

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

Roger T. Stevens and Christopher D. Watkins



Featuring 32 pages of
full-color graphics you can
create on your computer.

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

Roger T. Stevens and Christopher D. Watkins

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

Roger T. Stevens and Christopher D. Watkins

**M&T Books**

A Division of M&T Publishing, Inc.
501 Galveston Drive
Redwood City, CA 94063

© 1991 by M&T Publishing, Inc.

Printed in the United States of America

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage and retrieval system, without prior written permission from the Publisher. Contact the Publisher for information on foreign rights.

Limits of Liability and Disclaimer of Warranty

The Author and Publisher of this book have used their best efforts in preparing the book and the programs contained in it. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness.

The Author and Publisher make no warranty of any kind, expressed or implied, with regard to these programs or the documentation contained in this book. The Author and Publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.

Library of Congress Cataloging in Publication Data

Stevens, Roger T., 1927-

Graphics programming in Turbo Pascal/ by Roger T. Stevens and Christopher D. Watkins.
p. cm.

Includes bibliographical references and index.

ISBN 1-55851-131-8

1. Computer graphics. 2. Turbo Pascal (Computer program)

I. Watkins, Christopher D. II. Title.

T385.S77 1991

006.6'765—dc20

91-13214
CIP

Trademarks:

All products, names, and services are trademarks or registered trademarks of their respective companies.

Cover Design: Lauren Smith Design

94 93 92 91

5 4 3 2 1

Dedication

To our families and friends.

Many thanks to all of you for your consideration and support while we were writing this book. Without your help and understanding, this collection of programs and explanations would never have become a book.

Dedication

To our families and friends, who have always been our source of strength and support. We are grateful for your love and encouragement, which have helped us overcome all our challenges. This book is dedicated to you, as it is your support that has made our journey possible.

We hope that this book will provide you with a new perspective on life and help you find the answers to your questions. We are grateful for your love and support, and we hope that this book will be a source of inspiration for you.

We are grateful for your love and support, and we hope that this book will be a source of inspiration for you. We are grateful for your love and support, and we hope that this book will be a source of inspiration for you.

We are grateful for your love and support, and we hope that this book will be a source of inspiration for you. We are grateful for your love and support, and we hope that this book will be a source of inspiration for you.

We are grateful for your love and support, and we hope that this book will be a source of inspiration for you. We are grateful for your love and support, and we hope that this book will be a source of inspiration for you.

Contents

WHY THIS BOOK IS FOR YOU	1
INTRODUCTION	3
The Mathematics Module	4
The Graphics Interface Module	4
Using the Mathematics and Graphics Modules	5
Solid Modeling Theory and Data Base Structure	5
Adding Objects to the Scene	6
Sorting and Displaying Objects in a Scene	6
The Solid Modeling Program	6
The Description File Maker	7
Creating Data for a Solid Model	7
Z-Buffering	7
Ray Casting and Ray Tracing	8
Hardware and Software Required	8
Summing it Up	9
 PART I: UNIVERSAL ROUTINES	
 CHAPTER 1: THE MATHEMATICS MODULE	17
Mathematical Functions	17
The <i>Radians</i> and <i>Degrees</i> Functions	31
The <i>CosD</i> and <i>SinD</i> Functions	31
The <i>Power</i> Function	32
The <i>Log</i> Function	32
The <i>Exp10</i> Function	32
The <i>Sign</i> and <i>IntSign</i> Functions	32
The <i>IntSqrt</i> Function	32
The <i>IntPower</i> Function	33

Vector and Matrix Routines	33
The <i>Vec</i> and <i>VecInt</i> Procedures	33
The <i>UnVec</i> and <i>UnVecInt</i> Procedures	34
The <i>VecDot</i> Function	34
The <i>VecCross</i> Procedure	34
The <i>VecLen</i> Function	35
The <i>VecNormalize</i> Procedure	35
The <i>VecMatxMult</i> Procedure	35
The <i>VecSub</i> and <i>VecSubInt</i> Procedures	35
The <i>VecAdd</i> and <i>VecAdd3</i> Procedures	36
The <i>VecCopy</i> Procedure	36
The <i>VecLinComb</i> Procedure	36
The <i>VecScalMult</i> and <i>VecScalMultInt</i> Procedure	36
The <i>VecAddScalMult</i> Procedure	36
The <i>VecNull</i> and <i>VecNullInt</i> Procedures	37
The <i>VecElemMult</i> Procedure	37
Affine Transformation Routines	37
The <i>ZeroMatrix</i> Procedure	37
The <i>Translate3D</i> Procedure	37
The <i>Scale3D</i> Procedure	38
The <i>Rotate3D</i> Procedure	38
The <i>Multiply3DMatrices</i> Procedure	39
The <i>PrepareMatrix</i> Procedure	39
The <i>PrepareInvMatrix</i> Procedure	39
The <i>Transform</i> Procedure	40
 CHAPTER 2: THE GRAPHICS INTERFACE MODULE	41
The <i>SetMode</i> Procedure	41
The <i>PreCalc</i> Procedure	41
The <i>Plot</i> Procedure	41
The <i>ClearPalette</i> Procedure	57
The <i>SetPalette</i> Procedure	57

The <i>InitPalette1</i> Procedure	58
The <i>InitPalette2</i> Procedure	58
The <i>Cycle Palette</i> Procedure	59
The <i>Swap</i> Procedure	59
The <i>Circle</i> Procedure	59
The <i>Draw</i> Procedure	60
The <i>InitGraphics</i> Procedure	61
The <i>WaitForKey</i> Procedure	61
The <i>ExitGraphics</i> Procedure	61
The <i>Title</i> Procedure	61
Three-Dimensional Plotting Routines	62
The <i>InitPlotting</i> Procedure	62
The <i>InitPerspective</i> Procedure	62
The <i>MapCoordinates</i> Procedure	63
The <i>CartesianPlot3D</i> Procedure	63
The <i>CylindricalPlot3D</i> Procedure	64
The <i>SphericalPlot3D</i> Procedure	65
The <i>DrawLine3D</i> Procedure	65
The <i>PutPixel</i> Procedure	66
The <i>GetPixel</i> Function	66
Displaying the Coordinate Axes and the Color Palette	66

CHAPTER 3: HOW TO USE THE MODULES

Walking About the Mandelbrot Set	67
Simulating Two Orbiting Particles	77
The <i>Orbit-2P.Pas</i> Program	80
The <i>Orbit-3P.Pas</i> Program	85
Generating a Sierpinski Triangle	91
The Bifurcation Diagram	94
Creating a Strange Attractor	97
Generating the Lorenz Attractor	100

PART II: WIRE FRAME AND SOLID MODELING

CHAPTER 4: SOLID MODELING THEORY AND DATABASE

STRUCTURE.....	107
Scale Factors	108
Vertex and Facet Arrays	108
Loading and Saving <i>Vertex</i> and <i>Facet</i> Data.....	116
Drawing a Wire Frame Facet	118
Painting a Solid Facet on the Screen	118
A Manually Generated Data File Sample	118

CHAPTER 5: ADDING OBJECTS TO THE SCENE.....121

Initializing the Object Buffer	122
Inserting Object Information into the Object Buffer	122
Adding Edge Reflectors to a Scene	134
Adding Objects to a Scene from a Procedure	135
Adding Objects to a Scene From a Disk File	136

CHAPTER 6: SORTING AND DISPLAYING OBJECTS ON THE SCREEN.....137

Sorting Objects for Placement on the Screen	137
Placing an Object on the Screen	152
Displaying Objects and Reflections	153

CHAPTER 7: THE MODEL. PAS DESCRIPTION

FILE MAKER.....	155
------------------------	------------

CHAPTER 8: THE SOLID MODELING PROGRAM.....159

Viewer and Light Source Vector	160
Loading the Description File and Scene Data	177
Affine Transformations	178

Finding the Surface Normal Vector	179
The Illumination Model	179
Testing a Facet for Visibility	180
The Reflection Screen Buffer	180
Obtaining Facet Screen Coordinates	181
Painting a Solid Facet on the Screen	181
The <i>Model.Pas</i> Program	182

CHAPTER 9: CREATING OBJECT DATABASES183

Adding Vertices	183
Initializing Before Making Vertices	184
Creating Objects with the <i>MakeObj.Pas</i> Program	188
Generating a Cone or Pyramid Data File	191
Generating the Data File for a Cylinder	195
Generating the Data File for a Hemisphere	199
Generating the Data File for a Sphere	203
Generating Data Files for Equations	207
Generating the Data File for a Toroid	212
Generating Data Files for Solids of Revolution	216

CHAPTER 10: CREATING A SCENE FILE227

Using the <i>ScnMaker.Pas</i> Program	227
Keyboard Response Routines	232
The <i>ScnMaker.Pas</i> Program	233

PART III: Z BUFFERING AND HORIZON RENDERING

CHAPTER 11: Z BUFFERING THEORY AND DATABASE

STRUCTURE.....	261
Scale Factors	248

Clearing, Loading, and Saving Height Data	248
Getting an Object File	249
Displaying the Object Color Name	249

CHAPTER 12: THE RENDER.PAS DESCRIPTION

FILE MAKER.....	251
Keyboard Response Routines	251
The Main Program	251

CHAPTER 13: THE Z-BUFFER RENDERING PROGRAM.....261

Screen Procedures	261
The Illumination Model	277
Loading the Description Data	278
The <i>Render.Pas</i> Program	279
Displaying the Height Field	279
Placing Add-Ons on the Screen	280
Julia Set Images	281
Creating a Blended Sky	289

CHAPTER 14: CREATING AND WORKING WITH Z-BUFFER DATABASES

.....291	291
Generating the Database for a Hemisphere	291
Generating the Database for a Hemitoroid	291
Generating the Database for Equations	295
Generating a Magnetic Field Database	299
Finding the Highest and Lowest Points in the Height Array	305
Viewing a Slice of the Database	305
Removing Discontinuous Zeroes	314

CHAPTER 15: FRACTAL PROGRAMS TO GENERATE DATABASES	317
Generating Fractal Mountains	317
Three-Dimensional Mandelbrot Sets	323
Three-Dimensional Julia Sets	325
Fractals Using Quaternions	330
Quaternion Mathematics	335
Generating Quaternion Fractal Databases	336
Mountains from Plasmas	348
 PART IV: RAY TRACING	
CHAPTER 16: RAY TRACING FUNDAMENTALS	355
CHAPTER 17: HIGH-RESOLUTION GRAPHICS	359
CHAPTER 18: DEFINING A SCENE: THE .RT FILE	393
Formatting Constraints	393
CHAPTER 19: THE RAY TRACING PROGRAM	425
Ray-tracing overview	425
Loading an .RT File	482
Initializing the Noise Function	485
Scanning the Scene	486
Tracing the Ray	486
Determining the Color	491
Creating a Textured Surface	493
Completing the Ray Trace	493
CHAPTER 20: DISPLAYING A RAY-TRACED FILE	495
Creating a VGA Color Display	495

APPENDIX A: SOLID MODELING SCENE DEFINITION FILES.....	505
APPENDIX B: THREE PARTICLE ORBITS.....	523
APPENDIX C: MATERIALS USED IN RAY-TRACING.....	525
BIBLIOGRAPHY.....	531
INDEX	

Acknowledgments

All of the software in this book was written in Turbo Pascal, Versions 5.5 and 6.0, furnished by Borland International, 1800 Green Hills Rd., Scotts Valley, CA 95066.

Computer graphics, including high resolution 1024 x 768 pixel and 256-color displays were produced on Powergraph ERGO-VGA boards with 1 megabyte of memory, furnished by STB Systems, Inc., 1651 N. Glenville, Suite 210, Richardson, Texas 75081.

Thanks goes to Martin Jaspan for the ideas and initial BASIC code for the 3 particle orbit simulation program at the beginning of the book.

Why This Book is For You

If you have Turbo Pascal 5.0 or later compiler, an IBM PC or equivalent with a VGA card and a color monitor, perhaps you have tried your hand at producing some simple color displays. If you have, no doubt you were disappointed to find that not all of the graphics tools that you wanted were available and that you couldn't find enough information to produce really advanced displays. *Advanced Graphics Programming in Turbo Pascal* will provide you with the tools and information that you need to do some really advanced color graphics work.

Part I covers the tools needed to use advanced color graphics modes not covered by Turbo Pascal, including extended high-resolution VGA modes and 256 color graphics for highly realistic shading. Next, the mathematical functions and procedures needed to operate on vectors, including translation, rotation, and transformation from three-dimensional space to the two-dimensional screen are provided. Examples of the Mandelbrot and Julia sets, particles in orbit, a strange attractor, the Lorenz attractor, and a bifurcation diagram are provided to show how these tools are used to create interesting graphics displays.

Part II describes the theory and techniques required for the solid modeling of three-dimensional objects, including programs to display a number of primitive objects, and programs for creating scene files that can then be processed by the modeling program to create realistic scenes.

Part III covers the technique of z-buffering. This section includes programs to create data files for z-buffered rendering of scenes and includes information to render a number of interesting objects, including three-dimensional equations, and fractal Julia and dragon curves in the quaternions.

And finally, Part IV discusses ray-tracing. The programs needed to do this are all included, including the code necessary to create interesting object surface textures such as wood and marble.

Introduction

So you have a new VGA board and a color monitor to match and you want to produce some of those exciting new graphics displays that you've been reading about in the magazines. Your Turbo Pascal version 5.0 or greater has lots of graphics functions and procedures, but you're not quite sure how to put them together to produce something that will make people say "Oh!," when you bring it up on your screen. That is where this book comes in. We're going to show you how to create the mathematical and graphics functions that were left out of Turbo Pascal and then use them to create a number of interesting kinds of displays. You'll not only be able to generate all of the displays that are shown and described in this book, but the detailed descriptions of techniques and source code should give you some ideas for going off on your own and creating new displays that have never been done before.

The outline and all of the software in the book were written by Christopher D. Watkins. The book was written by Roger T. Stevens. The book is divided into four parts. Part I describes universal routines for mathematics and graphics. Part II will tell you how to do wire frame modeling and solid modeling. Wire frame modeling depicts three-dimensional objects as if they consisted of a frame of wires rather than a solid. It is a very fast way of drawing a three-dimensional scene on a two-dimensional display so that you can see the relative positions of objects and determine whether you have to reposition or resize any objects in order to have the picture the way you want it. Solid modeling actually projects the three-dimensional scene onto the screen, properly shading each facet of an object so that the scene looks like an artist's rendition. It requires a lot more computer time than wire frame modeling, so you will want to use it only when you're fairly sure that the picture has been defined the way you want it to be. Part III describes how to use the z-buffering and horizon rendering technique to create realistically lighted renditions of three-dimensional scenes. Part IV describes the use of ray-tracing and ray-casting techniques for creating realistically lighted pictures.

The Mathematics Module

Chapter 1 describes a number of mathematical functions and procedures that are useful in generating graphics scenes. They include a number of functions that were left out of Turbo Pascal but greatly simplify graphics programming. Also included are a full set of functions and procedures for creating and operating upon three-dimensional vectors. Such vectors are used to define and change the position of three-dimensional objects in space and to transform them on to a two-dimensional screen. Finally the module describes functions and procedures that create and operate upon four-dimensional matrices. If you want to take a three-dimensional vector or object and rotate it to a new orientation in space and possibly scale it to a new size at the same time, you will find that the transformation equations can be described by a 3×3 matrix. However, if you want a more general transformation that not only includes a rotation and scaling but also a linear translation in space, you need another operation on the vector or object. If this operation is included in a single matrix with the previously described operations, you will need a 4×4 matrix. The procedures described in this chapter include creating and initializing such matrices and performing various mathematical operations upon them.

The Graphics Interface Module

Chapter 2 describes a number of useful graphics routines. Included are a procedure for plotting a pixel to the screen, one for clearing the palette register, for setting the palette register, two functions for initializing the 256 colors of the 256 color display, procedures for drawing a circle or a line, a procedure to initialize the proper graphics mode, a procedure that causes the computer to wait for a key stroke, a procedure for exiting from the graphics mode, and one for setting up text screen colors. The chapter also includes a description of three dimensional and two-dimensional theory, orthogonal and perspective projection, and coordinate conversion. It also lists and describes the functions and procedures that are needed to perform these actions. Procedures that are used to draw a pixel to the screen and get a pixel from the screen are described as well as a procedure for drawing a wire outline for a facet of an object. The chapter also provides a procedure for placing the axes of the coordinate system and the current color palette on the screen and a procedure that permits toggling between displaying these on the screen, or not displaying them.

Using the Mathematics and Graphics Modules

Once you have assimilated the tools provided in the first two chapters, you're going to want to use them to create some interesting displays. Chapter 3 gives examples of a number of displays that are typical of those that you can create. The program for each display is listed and described, so that you can not only use it in its existing form, but will also know enough about the fundamentals so that you can modify it to meet your own requirements. The first program that is described begins by drawing a Mandelbrot set on one side of the screen. This is the most famous of all fractals. It is well-known that the Mandelbrot set is a sort of map of all Julia sets. The program is set up so that after the Mandelbrot set is drawn, a cursor appears. You can move the cursor around on the Mandelbrot set, and as you do, the corresponding Julia set for that particular set of coordinates is immediately drawn on the right-hand side of the screen.

The next two programs simulate two orbiting particles and three orbiting particles respectively. These particles actually appear to be rotating in orbit in space. The next display is a recursive triangle generator that creates representations of the well-known Sierpinski triangle. Next is a display of the bifurcation diagram. This shows how repeated iteration of the population growth equation for different parameter values can result in cases of settling to one stable solution, two stable solutions until chaos is encountered. This was one of the first discoveries that established the science of chaos. Another program creates a strange attractor, which is representative of a class of iterated equations where there is a particular set of curves to which the solutions are confined. The next program produces the Lorenz attractor, which is the first strange attractor that was discovered at the beginning of the development of chaos theory.

Solid Modeling Theory and Database Structure

Chapter 4 begins by describing the *Model.Inc* file. This file includes variable declarations for defining vertices and facets and *scale data* and *scale image* parameters for integer manipulation. Next, it provides routines to clear, load, and save vertex and facet data. Procedures are provided for drawing a wire facet and a solid facet. The chapter concludes by showing an example of how to create a file of data for a scene.

Adding Objects to the Scene

Chapter 5 describes and lists the procedures and functions that are needed to add objects to the object list data base. This includes initializing the object buffer, creating an object, rotating it, scaling it, translating it, defining its reflective characteristics, assigning it a color, and identifying whether it will reflect in another reflective object. A procedure is provided for setting a flag to indicate that the object should be sorted for order of placement. Procedures are given for adding edge reflectors to a scene. Finally, provisions are made for adding objects to the scene either from a .SCN disk file or from a procedure.

Sorting and Displaying Objects in a Scene

Assuming that we have some way of projecting objects onto a two-dimensional screen, it is important that we draw each object and its reflections in the proper order. If we construct the farthest objects first and the nearest objects last, parts of the nearer object may overlap parts of the farther objects, which is the way that the scene should be to achieve realism. If a far away object was drawn last, it might overlap nearer objects, which is not the way that the scene would appear when actually viewed. Consequently, Chapter 6 begins with functions and procedures that sort objects and their reflections to place them in the proper order for drawing to the screen. This chapter also includes a procedure that displays an object on the screen, in either vertex format, wire frame format, or solid model format. Another procedure displays objects and any reflections in the scene.

The Solid Modeling Program

Chapter 7 puts it all together by describing the complete solid modeling program, *Model.Pas*. This program begins by setting toggle switches that determine whether the scene should be displayed in the form of a vertex model, a wire frame model, or a solid model. It then performs sorting of the objects. It can also use affine transformations to change the size, position, and/or orientation of any object. It establishes the position of the viewer and the light source. Included is a procedure for obtaining the vector normal to the surface of an object facet. The program then runs an illumination model which determines the intensity of light on the facet. A test

is made as to whether the facet is visible, and the facets for objects and reflections, if visible, are finally projected to the screen to create the final picture. The program also includes the capability to load a .DES file which selects various options that determine what type of picture is created.

The Description File Maker

Chapter 8 describes the description file maker. This program creates a .DES description file from which the solid modeling program generates a scene. One purpose of this file is to specify whether the option should be selected that causes a complete color palette to be displayed at the side of the screen and the three axes of the Cartesian coordinate system to be displayed on the picture. The other option that is selected determines how the model will be displayed. Since generating a complete solid model is somewhat lengthy, two other options are available for faster drawing of the scene. These vertices are for simple display, or display of wire frame mode. Either of these options quickly creates a scene with sufficient detail for you to determine whether you have placed objects properly in the scene. Once you are satisfied with the placement, you can switch to the solid modeling option to create the final scene.

Creating Data for a Solid Model

The next two chapters describe how to create the data file from which a solid model scene is generated. Chapter 9 provides programs for generating such shapes as cones, cylinders, spheres, toroids, and solids of revolution. Chapter 10 describes a program for generating scenes and tells how to use it to create a data file with the description and placement of the objects that you want in the final scene. It is this data file that is used by the Model program to generate the final picture.

Z-Buffering

Part III of this book is concerned with the rendering of realistic scenes by use of the z-buffering technique. Chapter 11 discusses the theory of z-buffering and horizon rendering. Chapter 12 provides a z-buffering rendering program and describes how it works and how to use it. Chapter 13 describes the description file making program

which sets the options for the z-buffering rendering program. Chapter 14 describes the programs that are used to generate, examine, and modify a z-buffering database. Such a database contains the data that is used by the z-buffering rendering program to generate the final picture. Finally, Chapter 15 describes fractal programs that are used to create lunar mountains, quaternion dragons, and Mandelbrot and Julia sets using the continuous potential method.

Ray Casting and Ray Tracing

Part IV of this book is concerned with ray casting and ray tracing. Ray tracing is a technique for drawing a realistic picture of a three-dimensional scene on a two-dimensional display. It uses the computer to trace a path from the observer's eye to every pixel on the display screen and hence to whatever objects are intersected and then follows back to the original light sources. All of this information is combined to determine the color of each pixel at the display screen surface and this color is painted on the screen. Ray casting is somewhat simpler. It doesn't follow the ray to recreate all shadows and mirrored reflections but simply traces far enough to determine object colors. Chapter 16 describes the theory of ray tracing and describes the structure of the database used to store data for the process. Chapter 19 lists and describes the ray tracing program. In Chapter 18, we discuss database generators for ray-tracing programs. Chapter 19 tells how to ray-trace z-buffered objects as triangles. Chapter 17 describes the high-resolution 1024 x 768 x 256 color mode and its usage in the ray-tracing program.

Hardware and Software Required

All of the software in this book was written using Turbo Pascal version 5.5. The programs have also been checked out on the latest Turbo Pascal version 6.0 to assure that they are compatible. They may also run on earlier versions of Turbo Pascal, but you will need to make some changes in some of the functions and procedures. We have not tried to make the programs compatible with every Turbo Pascal version that is available. If you have one of the earlier versions that you came by legally, Borland International has a liberal upgrade policy that will enable you to get the latest Turbo Pascal version 6.0 at a reasonable cost. If you insist on using an earlier version or a pirated one, you're on your own.

These programs should run on any IBM PC compatible computer which has a VGA card and VGA monitor. Most of the software uses display mode 19, which is 320-by-200 pixels by 256 colors. With 256 colors, we can produce scenes that give quite a good representation of reality. We prefer super VGA cards which can produce 256 colors at higher resolutions. Unfortunately, the techniques for using these cards are not yet standardized and we couldn't write a book for each different card that is on the market. We used a Powergraph VGA card from STB and a Vega 1024i card from Video Seven in checking the software. We also include a 1024-by-768 pixel by 256 color mode using the PowerGraph card as an alternate in the ray-tracing programs. If you have another type of super-VGA card, you should be able to use what you have learned from this book to modify the programs to work with your own super-VGA card. You'll find that using a 286- or 386-based computer with a math coprocessor will make your life a lot more pleasant by speeding up the computing process, but with adequate patience, any of the programs will run on a slower machine.

Summing it Up

If you're like most Turbo Pascal programmers, you've been satisfied to use your compiler for fairly mundane tasks, such as routine calculations or simple procedures. Even if you have a VGA, you've seen it's power mainly through good quality characters during word processing or a fine solitaire game using Microsoft Windows 3.0. After trying the programs in this book, you'll be amazed at the graphics capabilities that lay hidden in your computer. These new graphics applications may inspire you to generate some new ones of your own and to investigate fields that have never been looked at before. After all, self-similar fractal curves lay dormant for nearly a hundred years because they required the power of modern computers for their investigation. It wasn't until Mandelbrot applied computer power to them that they came into their own. There are probably other areas of mathematics that are not new, but can be brought into new light with computer graphics power. Why don't you try to find some?

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

The following table presents a summary of the programs discussed in this book. For each program in the book, the following information is given:

- which chapter the program is shown in
- the name of the program or module (a module is code that does not stand alone, but is meant to be included in another program).
- a synopsis or brief description of what the program does
- a list of the modules which the program uses or includes within it
- a list of the input files required by the program. These files can be specific (e.g. "*Math.inc*"), or more generic (such as "**.des*").
- a similar list of the output files produced by the program.

Ch.	Program/Module	Synopsis	Includes	Input Files	Output Files
1	<i>Math.inc</i>	Basic mathematical functions, used by subsequent programs.	none	none	none
2	<i>Graph.inc</i>	Basic graphics functions, used by subsequent programs	none	none	none
3	<i>MSetJSet.pas</i>	Browse Mandelbrot set, display Julia set	<i>Math.inc</i> <i>Graph.inc</i>	none	none
3	<i>Orbit2P.pas</i>	Simulator for 2 Particle Orbit	<i>Math.inc</i> <i>Graph.inc</i>	none	none
3	<i>Orbit3P.pas</i>	Simulator for 3 Particle Orbit	<i>Math.inc</i> <i>Graph.inc</i>	none	none
3	<i>Triangle.pas</i>	Generate a Sierpinski Triangle	<i>Math.inc</i> <i>Graph.inc</i>	none	none
3	<i>Bifurcat.pas</i>	Generate a Bifurcation Diagram	<i>Math.inc</i> <i>Graph.inc</i>	none	none
3	<i>3DStrAttr.pas</i>	Generate a 3-D Strange Attractor	<i>Math.inc</i> <i>Graph.inc</i>	none	none
3	<i>3DLorenz.pas</i>	Generate a 3-D Lorenz Attractor	<i>Math.inc</i> <i>Graph.inc</i>	none	none

Ch.	Program/Module	Synopsis	Includes	Input Files	Output Files
4	<i>Model.inc</i>	Vertex and facet routines, used by subsequent programs	none	none	none
5	<i>AddObjs.inc</i>	Routines to add objects from a file	none	<i>grid.dat</i> <i>sphere.dat</i> <i>*.scn</i>	none
6	<i>DispObjs.inc</i>	sort and display objects on screen, used by subsequent routines	none	none	none
7	<i>DesMaker.pas</i>	creates a description (.des) file	none	user input	<i>Model.des</i>
8	<i>Model.pas</i>	solid modeling program	<i>Math.inc</i> <i>Graph.inc</i> <i>Model.inc</i> <i>AddObjs.inc</i> <i>DispObjs.inc</i>	<i>Model.des</i> <i>*.scn file</i>	outputs to screen display
9	<i>ShpMk.inc</i>	routines used to assist in creating object data files (.dat files), included by subsequent program	none	none	none
9	<i>MakeObj.pas</i>	creates an object data file (.dat file)	<i>Math.inc</i> <i>Graph.inc</i> <i>Model.inc</i> <i>ShpMk.inc</i>	user input	<i>*.dat</i>
9	<i>ConePyrm.pas</i>	generates object data files for a cone and a pyramid	<i>Math.inc</i> <i>Graph.inc</i> <i>Model.inc</i> <i>ShpMk.inc</i>	user input	<i>Pyramid.dat</i> <i>Cone.dat</i>
9	<i>Cylinder.Pas</i>	generates the object data file for a cylinder	<i>Math.inc</i> <i>Graph.inc</i> <i>Model.inc</i> <i>ShpMk.inc</i>	user input	<i>Cylinder.dat</i>
9	<i>HmSphere.pas</i>	generates the object data file for a hemisphere	<i>Math.inc</i> <i>Graph.inc</i> <i>Model.inc</i> <i>ShpMk.inc</i>	user input	<i>HmSphere.dat</i>
9	<i>Sphere.pas</i>	generates the object data file for a sphere	<i>Math.inc</i> <i>Graph.inc</i> <i>Model.inc</i> <i>ShpMk.inc</i>	user input	<i>Sphere.dat</i>

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

Ch.	Program/Module	Synopsis	Includes	Input Files	Output Files
9	<i>PlotEqns.pas</i>	generates the object data file for a grid and for 4 different equations	<i>Math.inc</i> <i>Graph.inc</i> <i>Model.inc</i> <i>ShpMk.inc</i>	user input	<i>Grid.dat</i> <i>PlotEqn1.dat</i> <i>PlotEqn2.dat</i> <i>PlotEqn3.dat</i> <i>PlotEqn4.dat</i>
9	<i>Toroid.pas</i>	generates the object data file for a toroid	<i>Math.inc</i> <i>Graph.inc</i> <i>Model.inc</i> <i>ShpMk.inc</i>	user input	<i>Toroid.dat</i>
9	<i>SolOfRev.pas</i>	generates the object data file for a number of solids of revolution	<i>Math.inc</i> <i>Graph.inc</i> <i>Model.inc</i> <i>ShpMk.inc</i>	user input	<i>Beads.dat</i> <i>LgBeads.dat</i> <i>Funnels.dat</i> <i>MechPart.dat</i> <i>Rocket.dat</i> <i>ChesPawn.dat</i>
10	<i>ScnMaker.pas</i>	generates scene (.scn) files	none	user input	*.scn
11	<i>Render.inc</i>	routines for file and structure of z-buffering program, included by subsequent program	none	none	none
12	<i>DesMake.pas</i>	creates a description (.des) file to be used by <i>Render.pas</i> program	<i>Render.inc</i>	user input	*.des
13	<i>Render.pas</i>	z-buffer rendering program	<i>Math.inc</i> <i>Graph.inc</i> <i>Render.inc</i> <i>AddOns.inc</i>	*.dat *.des <i>CPM1</i> <i>CPM2</i> <i>CPMJ1</i> <i>CPMJ2</i>	
13	<i>AddOns.inc</i>	included by z-buffer rendering program (<i>Render.pas</i>)	none	none	none
14	<i>HmSphere.pas</i>	generates a z-buffer database for painting a hemisphere on the screen	<i>Render.inc</i> <i>Math.inc</i>		<i>HmSphere.dat</i>
14	<i>HmToroid.pas</i>	generates a z-buffer database for painting a hemitoroid on the screen	<i>Render.inc</i> <i>Math.inc</i>		<i>HmToroid.dat</i> <i>Bowl.dat</i>
14	<i>PlotEqn.pas</i>	generates a z-buffer database for drawing 4 different equations	<i>Render.inc</i> <i>Math.inc</i>		<i>PlotEqn1.dat</i> <i>PlotEqn2.dat</i> <i>PlotEqn3.dat</i> <i>PlotEqn4.dat</i>
14	<i>MagnFlds.pas</i>	generates a z-buffer database for drawing the image of magnetic fields	<i>Render.inc</i> <i>Math.inc</i>		<i>MagnFlds.dat</i>

Ch.	Program/Module	Synopsis	Includes	Input Files	Output Files
14	<i>FndLimits.pas</i>	program to find highest and lowest z-values	<i>Render.inc</i>	*.dat	outputs to screen
14	<i>ViewSlic.pas</i>	program to view a cross-section of displayed data	<i>Math.inc</i> <i>Graph.inc</i> <i>Render.inc</i>	*.dat	draws to screen
14	<i>Smooth.pas</i>	program to correct erroneous or discontinuous points	<i>Render.inc</i>	*.dat	corrected file
15	<i>Moutain.pas</i>	generates fractal mountain databases	<i>Math.inc</i> <i>Graph.inc</i> <i>Render.inc</i>		<i>mountain.dat</i>
15	<i>CPM.pas</i>	creates data file for 3-D Mandelbrot set	<i>Math.inc</i> <i>Graph.inc</i> <i>Render.inc</i>		<i>CPM1.dat</i> <i>CPM2.dat</i>
15	<i>CPMJ.pas</i>	creates data file for 3-D Julia set	<i>Math.inc</i> <i>Graph.inc</i> <i>Render.inc</i>		<i>CPMJ1.dat</i> <i>CPMJ2.dat</i>
15	<i>Quater.inc</i>	math routines used by quaternion file generator	none	none	none
15	<i>Quater.pas</i>	generates data file for quaternion fractals	<i>Math.inc</i> <i>Render.inc</i> <i>Quater.inc</i>		<i>dragon.dat</i> <i>revsolid.dat</i>
17	<i>Math2.inc</i>	file from chapter 1 modified for super VGA	none	none	none
17	<i>Graph2.inc</i>	file from chapter 2 modified for super VGA	none	none	none
18	<i>Example1.rt</i>	example text data files used by ray tracing program in Chapter 19 (<i>RayTrace.pas</i>). Note: these are data files, not programs.	none	none	none
19	<i>RayTrace.pas</i>	recursive ray tracing program	<i>Math2.inc</i> <i>Graph2.inc</i>	*.rt files	*.cpr files
20	<i>Display.pas</i>	using a color histogram, display the .cpr file created by ray tracing program	<i>Math2.inc</i> <i>Graph2.inc</i>	*.cpr	screen display

Part I

CHAPTER 1

The Mathematics Module

UNIVERSAL ROUTINES

CHAPTER 1

The Mathematics Module

Before you get into the actual creation of images with Turbo Pascal, you'll need to develop mathematical functions that you can use repeatedly throughout the graphics programs. All of these procedures are included in a module called *Math.Inc*, in each graphics program. There are three types of mathematical functions included in the module: Functions that manipulate numbers, vector and matrix functions, and affine transformations, which transform the position of points using a four-by-four matrix. All three types of functions listed in Figure 1-1 are described in detail in the following sections.

Mathematical Functions

The first thing you need to do is create a number of mathematical functions not included in Turbo Pascal. Before starting with the functions, the definitions of three constants are required: *Ln10*, the natural logarithm of 10 (2.3025851), used in converting between natural logarithms and logarithms to the base 10; *PiOver180* (0.0174533); and, *PiUnder180* (57.2957795). (Logarithms to the base 10 are more normally used in algebra and trigonometry classes, but the only logarithmic function native to Turbo Pascal is that for finding the natural logarithm—to the base *e*. You need the constant *Ln10* to convert from the normal logarithms to ones understood by Turbo Pascal.)

{

Mathematical Functions

Radians - converts degrees to radians
Degrees - converts radians to degrees

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

CosD - cosine in degrees
SinD - sine in degrees
Power - power a^n
Log - log base 10
Exp10 - exp base 10
Sign - negative=-1 positive=1 null=0
IntSign - negative=-1 positive=1 null=0
IntSqrt - integer square root
IntPower - integer power a^n

}

Const

Ln10 = 2.30258509299405E+000; { Ln(10) = 2.3025851 }
PiOver180 = 1.74532925199433E-002; { Pi / 180 = 0.0174533 }
PiUnder180 = 5.72957795130823E+001; { 180 / Pi = 57.2957795 }

Function Radians(Angle : Real) : Real;

Begin

 Radians := Angle * PiOver180 { Angle * Pi / 180 }

End;

Function Degrees(Angle : Real) : Real;

Begin

 Degrees := Angle * PiUnder180 { Angle * 180 / Pi }

End;

Function CosD(Angle : Real) : Real;

Begin

 CosD := Cos(Radians(Angle))

End;

Function SinD(Angle : Real) : Real;

THE MATHEMATICS MODULE

```
Begin
  SinD := Sin( Radians( Angle ) )
End;
```

```
Function Power(Base : Real; Exponent : Integer) : Real;
```

```
  Var
    BPower : Real;
    t      : Integer;
```

```
  Begin
    If (Exponent = 0) Then
      Power := 1
    Else
      Begin
        BPower := 1.0;
        For t := 1 To Exponent Do
          BPower := BPower * Base;
        Power := BPower
      End
    End;
  End;
```

```
Function Log(x : Real) : Real;
```

```
  Begin
    Log := Ln(x) / Ln10
  End;
```

```
Function Exp10(x : Real) : Real;
```

```
  Begin
    Exp10 := Exp( x * Ln10 )
  End;
```

```
Function Sign(x : Real) : Integer;
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
Begin
  If (x < 0) Then
    Sign := -1
  Else
    If (x > 0) Then
      Sign := 1
    Else
      Sign := 0
End;
```

Function IntSign(x : Integer) : Integer;

```
Begin
  If (x < 0) Then
    IntSign := -1
  Else
    If (x > 0) Then
      IntSign := 1
    Else
      IntSign := 0
End;
```

Function IntSqrt(x : Integer) : Integer;

```
Var
  OddInt, OldArg, FirstSqrt : Integer;
```

```
Begin
  OddInt := 1;
  OldArg := x;
  While x >= 0 Do
    Begin
      x := x - OddInt;
      OddInt := OddInt + 2
    End;
  FirstSqrt := OddInt Shr 1;
  If (Sqr(FirstSqrt) - FirstSqrt + 1 > OldArg) Then
    { regulate overshoot }
```


THE MATHEMATICS MODULE

```

      IntSqrt := FirstSqrt - 1
    Else
      IntSqrt := FirstSqrt
    End;

```

```

Function IntPower(Base, Exponent : Integer) : Integer;

```

```

Begin
  If Exponent = 0 Then
    IntPower := 1
  Else
    IntPower := Base * IntPower(Base, Exponent - 1)
  End;

```

```

{

```

Vector and Matrix Routines

Vec	- Make Vector	
VecInt	- Make Integer Vector	
UnVec	- Get Components of Vector	
UnVecInt	- Get Components of Integer Vector	
VecDot	- Vector Dot Product	$P = A \cdot B$
VecCross	- Vector Cross Product	$C = A \times B$
VecLen	- Vector Length	$L = A $
VecNormalize	- Vector Normalize	$B = A / A $
VecMatxMult	- Vector Matrix Multiply	$B = \text{Matx} \times A$
VecSub	- Vector Subtraction	$C = A - B$
VecSubInt	- Vector Subtraction Integer	$C = A - B$
VecAdd	- Vector Addition	$C = A + B$
VecAdd3	- Vector Addition	$D = A + B + C$
VecCopy	- Vector Copy	$B = A$
VecLinComb	- Vector Linear Combination	$C = rA + sB$
VecScalMult	- Vector Scalar Multiple	$B = rA$
VecScalMultI	- Vector Scalar Multiple	$B = rA,$
		$A \text{ is integer}$
VecScalMultInt	- Vector Scalar Multiple and Rounding	$B = \text{Int}(rA)$

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

VecAddScalMult	- Vector Add Scalar Multiple	$C = rA + B$
VecNull	- Vector Null	$C = 0$
VecNullInt	- Vector Null Integer	$C = 0$
VecElemMult	- Vector Element Multiply	$C = r * A * B$

}

Type

```
TDA    = Array[0..2] Of Real;  
TDIA   = Array[0..2] Of Integer;  
FDA    = Array[0..3] Of Real;  
Matx4x4 = Array[0..3, 0..3] Of Real;
```

```
Procedure Vec(r, s, t : Real; Var A : TDA);  
    { Make Vector A = ri + sj + tk }
```

```
Begin
```

```
    A[0] := r;  
    A[1] := s;  
    A[2] := t
```

```
End;
```

```
Procedure VecInt(r, s, t : Integer; Var A : TDIA);  
    { Make Integer Vector A = ri + sj + tk }
```

```
Begin
```

```
    A[0] := r;  
    A[1] := s;  
    A[2] := t
```

```
End;
```

```
Procedure UnVec(A : TDA; Var r, s, t : Real);  
    { Get Components of Vector }
```

```
Begin
```

```
    r := A[0];  
    s := A[1];  
    t := A[2]
```

```
End;
```


THE MATHEMATICS MODULE

```
Procedure UnVecInt(A : TDIA; Var r, s, t : Integer);
```

```
  { Get Components of Integer Vector }
```

```
Begin
```

```
  r := A[0];
```

```
  s := A[1];
```

```
  t := A[2]
```

```
End;
```

```
Function VecDot(A, B : TDA) : Real;
```

```
  { Vector Dot Product =  $A \cdot B$  }
```

```
Begin
```

```
  VecDot := A[0]*B[0] + A[1]*B[1] + A[2]*B[2]
```

```
End;
```

```
Procedure VecCross(A, B : TDA; Var C : TDA);
```

```
  { Vector Cross Product  $C = A \times B$  }
```

```
Begin
```

```
  C[0] := A[1]*B[2] - A[2]*B[1];
```

```
  C[1] := A[2]*B[0] - A[0]*B[2];
```

```
  C[2] := A[0]*B[1] - A[1]*B[0]
```

```
End;
```

```
Function VecLen(A : TDA) : Real;
```

```
  { Vector Length =  $|A|$  }
```

```
Begin
```

```
  VecLen := Sqrt( Sqr(A[0]) + Sqr(A[1]) + Sqr(A[2]) )
```

```
End;
```

```
Procedure VecNormalize(Var A : TDA);
```

```
  { Vector Normalize  $B = A / |A|$  }
```

```
Var
```

```
  dist, invdist : Real;
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
Begin
  dist := VecLen(A);
  If Not(dist = 0.0) Then
    Begin
      invdist := 1 / dist;
      A[0] := A[0] * invdist;
      A[1] := A[1] * invdist;
      A[2] := A[2] * invdist;
    End
  Else
    Begin
      WriteLn('Zero-Length Vectors cannot be Normalized');
      Halt
    End
  End;
End;
```

```
Procedure VecMatxMult(A : FDA; Matrix : Matx4x4; Var B : FDA);
{ Vector Matrix Multiply B = Matx x A }
Var
  mRow, mCol : Integer;

Begin
  For mCol := 0 To 3 Do
    Begin
      B[mCol] := 0;
      For mRow := 0 To 3 Do B[mCol] := B[mCol] + A[mRow] *
        Matrix[mRow,mCol]
      End
    End
  End;
End;
```

```
Procedure VecSub(A, B : TDA; Var C : TDA);
{ Vector Subtraction C = A - B }

Begin
  C[0] := A[0] - B[0];
  C[1] := A[1] - B[1];
  C[2] := A[2] - B[2]
End;
```


THE MATHEMATICS MODULE

```
Procedure VecSubInt(A, B : TDIA; Var C : TDA);  
  { Vector Subtraction Integer to Real  $C = A - B$  }
```

```
  Begin
```

```
    C[0] := A[0] - B[0];
```

```
    C[1] := A[1] - B[1];
```

```
    C[2] := A[2] - B[2]
```

```
  End;
```

```
Procedure VecAdd(A, B : TDA; Var C : TDA);  
  { Vector Addition  $C = A + B$  }
```

```
  Begin
```

```
    C[0] := A[0] + B[0];
```

```
    C[1] := A[1] + B[1];
```

```
    C[2] := A[2] + B[2]
```

```
  End;
```

```
Procedure VecAdd3(A, B, C : TDA; Var D : TDA);  
  { Vector Addition  $D = A + B + C$  }
```

```
  Begin
```

```
    D[0] := A[0] + B[0] + C[0];
```

```
    D[1] := A[1] + B[1] + C[1];
```

```
    D[2] := A[2] + B[2] + C[2]
```

```
  End;
```

```
Procedure VecCopy(A : TDA; Var B : TDA);  
  { Vector Copy  $B = A$  }
```

```
  Begin
```

```
    B := A
```

```
  End;
```

```
Procedure VecLinComb(r : Real; A : TDA; s : Real; B : TDA;  
                    Var C : TDA);
```

```
  { Vector Linear Combination  $C = rA + sB$  }
```

```
  Begin
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
    C[0] := r * A[0] + s * B[0];  
    C[1] := r * A[1] + s * B[1];  
    C[2] := r * A[2] + s * B[2]  
End;
```

```
Procedure VecScalMult(r : Real; A : TDA; Var B : TDA);  
    { Vector Scalar Multiple B = rA }  
Begin  
    B[0] := r * A[0];  
    B[1] := r * A[1];  
    B[2] := r * A[2]  
End;
```

```
Procedure VecScalMultI(r : Real; A : TDIA; Var B : TDA);  
    { Vector Scalar Multiple B = rA, A is integer }  
Begin  
    B[0] := r * A[0];  
    B[1] := r * A[1];  
    B[2] := r * A[2]  
End;
```

```
Procedure VecScalMultInt(r : Real; A : TDA; Var B : TDIA);  
    { Vector Scalar Multiple and Rounding B = Int(rA) }  
Begin  
    B[0] := Round(r * A[0]);  
    B[1] := Round(r * A[1]);  
    B[2] := Round(r * A[2])  
End;
```

```
Procedure VecAddScalMult(r : Real; A, B : TDA; Var C : TDA);  
    { Vector Add Scalar Multiple C = rA + B }  
Begin  
    C[0] := r * A[0] + B[0];  
    C[1] := r * A[1] + B[1];  
    C[2] := r * A[2] + B[2]  
End;
```


THE MATHEMATICS MODULE

```
Procedure VecNull(Var A : TDA);  
  { Vector Null C = 0 }
```

```
Begin
```

```
  A[0] := 0.0;
```

```
  A[1] := 0.0;
```

```
  A[2] := 0.0
```

```
End;
```

```
Procedure VecNullInt(Var A : TDIA);  
  { Vector Null Integer C = 0 }
```

```
Begin
```

```
  A[0] := 0;
```

```
  A[1] := 0;
```

```
  A[2] := 0
```

```
End;
```

```
Procedure VecElemMult(r : Real; A, B : TDA; Var C : TDA);  
  { Vector Element Multiply C = r * A * B }
```

```
Begin
```

```
  C[0] := r * A[0] * B[0];
```

```
  C[1] := r * A[1] * B[1];
```

```
  C[2] := r * A[2] * B[2]
```

```
End;
```

```
{
```

Affine Transform Routines

ZeroMatrix	- zeros the elements of a 4x4 matrix
Translate3D	- make translation matrix
Scale3D	- make scaling matrix
Rotate3D	- make rotation matrix
ZeroAllMatrices	- zeros all matrices used in transformation
Multiply3DMatrices	- multiply two 4x4 matrices

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
PrepareMatrix      - prepare the transformation matrix  
                   (Tm = S*R*T)  
PrepareInvMatrix   - prepare the inverse transformation  
                   matrix  
Transform          - multiply a vertex by the transformation  
                   matrix  
  
}
```

```
Procedure ZeroMatrix(Var A : Matx4x4);
```

```
  Var  
    i, j : Integer;  
  
  Begin  
    For i := 0 To 3 Do  
      For j := 0 To 3 Do  
        A[i,j] := 0.0  
      End;  
    End;
```

```
Procedure Translate3D(tx, ty, tz : Real; Var A : Matx4x4);
```

```
  Var  
    i : Integer;  
  
  Begin  
    ZeroMatrix(A);  
    For i := 0 To 3 Do  
      A[i,i] := 1.0;  
      A[0,3] := -tx;  
      A[1,3] := -ty;  
      A[2,3] := -tz  
    End;
```

```
Procedure Scale3D(sx, sy, sz : Real; Var A : Matx4x4);
```

```
  Begin  
    ZeroMatrix(A);
```



```

A[0,0] := sx;
A[1,1] := sy;
A[2,2] := sz;
A[3,3] := 1.0
End;
```

```

Procedure Rotate3D(m : Integer; Theta : Real; Var A : Matx4x4);
```

```

Var
  m1, m2 : Integer;
  c, s    : Real;
```

```

Begin
  ZeroMatrix(A);
  A[m-1,m-1] := 1.0;
  A[3,3] := 1.0;
  m1 := (m Mod 3) + 1;
  m2 := (m1 Mod 3);
  m1 := m1 - 1;
  c := CosD( Theta );
  s := SinD( Theta );
  A[m1,m1] := c;
  A[m1,m2] := s;
  A[m2,m2] := c;
  A[m2,m1] := -s
End;
```

```

Procedure Multiply3DMatrices(A, B : Matx4x4; Var C : Matx4x4);
```

```

Var
  i, j, k : Integer;
  ab      : Real;
```

```

Begin
  For i := 0 To 3 Do
    For j := 0 To 3 Do
      Begin
        ab := 0;
        For k := 0 To 3 Do
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
        ab := ab + A[i,k] * B[k,j];  
        C[i,j] := ab  
    End  
End;
```

```
Procedure PrepareMatrix(Tx, Ty, Tz, Sx, Sy, Sz, Rx, Ry, Rz :  
                        Real; Var XForm : Matx4x4);
```

```
Var  
    M1, M2, M3, M4, M5, M6, M7, M8, M9 : Matx4x4;
```

```
Begin  
    Scale3D ( Sx, Sy, Sz, M1 );  
    Rotate3D ( 1, Rx, M2 );  
    Rotate3D ( 2, Ry, M3 );  
    Rotate3D ( 3, Rz, M4 );  
    Translate3D( Tx, Ty, Tz, M5 );  
    Multiply3DMatrices( M2, M1, M6 );  
    Multiply3DMatrices( M3, M6, M7 );  
    Multiply3DMatrices( M4, M7, M8 );  
    Multiply3DMatrices( M5, M8, M9 );  
    XForm := M9  
End;
```

```
Procedure PrepareInvMatrix(Tx, Ty, Tz, Sx, Sy, Sz, Rx, Ry, Rz :  
                           Real; Var XForm : Matx4x4);
```

```
Var  
    M1, M2, M3, M4, M5, M6, M7, M8, M9 : Matx4x4;
```

```
Begin  
    Scale3D ( Sx, Sy, Sz, M1 );  
    Rotate3D ( 1, Rx, M2 );  
    Rotate3D ( 2, Ry, M3 );  
    Rotate3D ( 3, Rz, M4 );  
    Translate3D( Tx, Ty, Tz, M5 );  
    Multiply3DMatrices( M4, M5, M6 );  
    Multiply3DMatrices( M3, M6, M7 );
```



```

Multiply3DMatrices( M2, M7, M8 );
Multiply3DMatrices( M1, M8, M9 );
XForm := M9
End;

Procedure Transform(A : TDA; M : Matx4x4; Var B : TDA);

Begin
  B[0] := M[0,0] * A[0] + M[0,1] * A[1] + M[0,2] * A[2] +
          M[0,3];
  B[1] := M[1,0] * A[0] + M[1,1] * A[1] + M[1,2] * A[2] +
          M[1,3];
  B[2] := M[2,0] * A[0] + M[2,1] * A[1] + M[2,2] * A[2] +
          M[2,3];
End;

```

Figure 1-1. The Mathematics Module

You usually specify angles in degrees, but the trigonometric functions native to Turbo Pascal specify angles in radians. If you want to use angles in degrees, you'll need a way to convert, which is provided through use of the next two constants.

The *Radians* and *Degrees* Functions

These are very simple. They use the constants *PiOver180* and *PiUnder180* and a multiplication to perform conversions between degrees and radians. The *Radians* function converts degrees to radians. The *Degrees* function works in reverse, converting radians to degrees.

The *CosD* and *SinD* Functions

These two functions are equally simple. The *CosD* function finds the cosine of an angle stated in degrees by converting the angle to radians. It then uses the Turbo Pascal *cos* function to find the angle's cosine. The *SinD* function similarly finds the sine of an angle specified in degrees by converting the angle to radians, and then using the Turbo Pascal *sin* function to find the angle's sine.

The *Power* Function

The *Power* function finds the value of a real (floating point) number raised to an integer power. It returns a one if the power is zero (any number raised to the zeroth power is one). Otherwise, the function begins by initializing the variable *BPower* to one. Then a *For* loop multiplies *BPower* by the floating-point number passed to the function as many times as the integer exponent, placing the new value in *BPower* at each iteration. When the loop ends, the desired result is in *BPower*, which is returned by the function.

The *Log* Function

The *Log* function finds the logarithm of a number to the base 10. It does this by first taking the natural logarithm of the number (using the Turbo Pascal function), and then dividing it by the previously defined constant *Ln10* (the natural logarithm of 10). The desired result is then returned by the function.

The *Exp10* Function

The *Exp10* function finds the value of ten raised to a real number power. The desired power is first multiplied by the natural logarithm of 10 and the result raised to the *e* power by using the *Exp* function of Turbo Pascal. The function then returns the output of the *Exp* function.

The *Sign* and *IntSign* Functions

These functions are used to find the sign of a real number and an integer, respectively: Either function returns an integer. This integer is set to -1 if the number is less than zero, +1 if the number is greater than zero, and 0 if the number is equal to zero.

The *IntSqrt* Function

The *IntSqrt* finds the square root of an integer. If the square root of the integer is in the form of a real number, with the decimal rounded off, the result is the same as the square root of the integer. The resulting integer is simply the result truncated if the decimal part is less than 0.5. If the decimal part is greater than or equal to 0.5,

the result is one greater than the truncated result. The function for doing this is a little obscure. You have to work out a few examples for yourself to see if it really works. It begins with a *For* loop that starts with *OddInt* equal to one, and *x* equal to the number whose square root is being calculated. At each iteration it reduces *x* by *OddInt* and then increases *OddInt* by 2. When *x* is less than zero, the loop terminates. At this point half of the value of *OddInt* is either the square root of the integer or one more than the square root of the integer. The function uses a shift to get half the *OddInt* value, and uses an *if* statement to determine whether the conditions are met for which the result should be decremented. Finally, the result is returned by the function.

The *IntPower* Function

The *IntPower* function is elegant in its use of recursion to raise an integer to an integer power. Recursion is the calling of a function within that same function. Not all computer languages have the capability to do this without becoming confused, but Turbo Pascal is written so that recurrence is permissible. Note that except for the case of a zero exponent, the function keeps calling itself at the next decrement of the power. Finally, it is decremented to zero, whereupon the number one is returned to the previous call. One is then multiplied by the base and returned to the next previous call, and so forth until the highest level is reached.

Vector and Matrix Routines

In order to move objects around in space and convert them to two-dimensional displays, you'll need to be able to manipulate vectors. Turbo Pascal has no vector operations, so you'll have to create your own. The vector portion of the mathematics module begins by defining the vector type *TDA*. It is an array of three real numbers representing the *x*, *y*, and *z* components of a three-dimensional vector. Similarly, the type *TDIA* is defined, which is a three-dimensional integer vector. An array *FDA* is defined, which is a four-dimensional vector of real numbers. Finally, the type *Matx4x4* is defined, which is a four-by-four element matrix.

The *Vec* and *VecInt* Procedures

The *Vec* and *VecInt* procedures create vectors. Three real numbers and a vector

of the real-number type pass to the procedure *Vec*. The procedure sets the three members of the array to the three, real, passed values. The *VecInt* procedure works similarly, except that three integers and a vector of the integer type are passed to the procedure *Vec*.

The *UnVec* and *UnVecInt* Procedures

These are the reverse of the procedures just described. The same three variables and a vector are passed, but the procedure transfers the three vector components to the three variables instead of the other way around.

The *VecDot* Function

The *VecDot* function computes the dot product of two, real-number, three-dimensional vectors. Remember from your math courses that the dot product of two vectors is a number, which is the sum of the products of corresponding elements of the two vectors. The value of the dot product is:

$$a \cdot b = AB \cos x \quad (\text{Equation 1-1})$$

where a and b are the vectors, A and B are their magnitudes, and x is the angle between them. The parameters passed to the function are two, real-number, three-dimensional vectors. The function returns a real number that is the vector dot product.

The *VecCross* Procedure

The *VecCross* procedure finds the cross-product of two, real-number, three-dimensional vectors. The vector cross-product has a length of:

$$|a \times b| = AB \sin x \quad (\text{Equation 1-2})$$

where A and B are the lengths of the vectors a and b respectively and x is the angle between the two vectors. The direction of the vector cross-product is perpendicular to the plane defined by the vectors a and b . The vector cross-product is:

$$a \times b = (a_y b_z - a_z b_y, a_z b_x - a_x b_z, a_x b_y - a_y b_x) \quad (\text{Equation 1-3})$$

The *VecLen* Function

The *VecLen* function finds the length of a vector.

The *VecNormalize* Procedure

A normalized vector is one that has the same direction as a given vector, but has a unit length of one. Since a zero length vector really has no direction, it is a degenerate case that cannot be normalized. Hence, the first thing that the *VecNormalize* function does is call *VecLen* to find the length of the vector. If this distance is zero the function returns after displaying an error message that says that the vector could not be normalized. Otherwise, the inverse of the vector length is found and each component of the vector is multiplied by it. This gives the components of a new vector having the same direction, but a length of one. This is therefore the normalization of the given vector. The procedure is designed in such a way that the normalized vector replaces the original vector which was passed to the procedure.

The *VecMatxMult* Procedure

The *VecMatxMult* procedure performs the multiplication of a vector by a four-by-four matrix. The operation is:

$$b = \text{Matx} \times a \quad (\text{Equation 1-4})$$

where a and b are real three-dimensional vectors and Matx is a four-by-four matrix. As indicated, the result of the multiplication is a vector. The procedure uses two nested *For* loops to multiply the appropriate elements. They then sum them into the new vector components. The vector a and the matrix are passed to the procedure and the result of the multiplication is returned in b .

The *VecSub* and *VecSubInt* Procedures

The *VecSub* procedure subtracts one real, three-dimensional vector from another and returns the result. Similarly, the *VecSubInt* procedure subtracts one-integer, three-dimensional vector from another and returns the result.

The *VecAdd* and *VecAdd3* Procedures

The *VecAdd* procedure adds one real three-dimensional vector to another and returns the result. Similarly, the *VecAdd3* procedure adds three-real, three-dimensional vectors and returns the result.

The *VecCopy* Procedure

The *VecCopy* procedure simply copies the contents of one vector to another.

The *VecLinComb* Procedure

The *VecLinComb* accepts two real numbers and two vectors as inputs. The first real number multiplies all elements of the first vector and the second real number multiplies all elements of the second vector. The resulting vectors are added. Thus the operation is:

$$c = Ra + Sb \quad (\text{Equation 1-5})$$

where a , b , and c are real, three-dimensional vectors and R and S are real numbers.

The *VecScalMult* and *VecScalMultInt* Procedure

The *VecScalMult* procedure multiplies a real, three-dimensional vector by a real number. Each element of the vector is multiplied by the real number to give a new vector. Similarly, the *VecScalMultInt* procedure multiplies an integer, three-dimensional vector by an integer. Each element of the vector is multiplied by the integer to give a new vector.

The *VecAddScalMult* Procedure

The *VecAddScalMult* procedure multiplies a real, three-dimensional vector by a real number and adds the result to another real, three-dimensional vector. The operation is:

$$c = R*a + b \quad (\text{Equation 1-6})$$

where a , b , and c are real, three-dimensional vectors and R is a real number.

The *VecNull* and *VecNullInt* Procedures

The *VecNull* procedure creates a zero or null real, three-dimensional vector, (i.e. one in which each element is zero). Similarly, the *VecNullInt* procedure creates a null integer, three-dimensional vector.

The *VecElemMult* Procedure

The *VecElemMult* procedure multiplies a real, three-dimensional vector by a real number and then multiplies the result by another real, three-dimensional vector. The operation is:

$$c = R * a * b \quad (\text{Equation 1-7})$$

where a , b , and c are real, three-dimensional vectors, and R is a real number.

Affine Transformation Routines

Affine transformation routines create and manipulate four-by-four matrices. They create matrices for translation, scaling, and rotation, combine these into a single transformation matrix, and perform various matrix mathematical operations.

The *ZeroMatrix* Procedure

The *ZeroMatrix* procedure simply zeroes all the elements of a four-by-four matrix.

The *Translate3D* Procedure

The *Translate3D* procedure creates the matrix for performing a linear translation of a vector to a new location in space. This matrix consists of ones, except for the last three elements of the first column, which are set to the negative of the three translation parameters passed to the procedure. This creates the following matrix:

$$T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -t_x & 1 & 0 & 0 \\ -t_y & 0 & 1 & 0 \\ -t_z & 0 & 0 & 1 \end{bmatrix}$$

(Equation 1-8)

The *Scale3D* Procedure

The *Scale3D* procedure leaves the origin of a real, three-dimensional vector at the same point with the direction unchanged, but changes the size of the vector. Scale factors for each of the three vector components are passed to the procedure. These components become the first three diagonal elements of the matrix; the last diagonal element is a one. All other elements of the matrix are set to zero. The result is the following matrix:

$$T = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

(Equation 1-9)

The *Rotate3D* Procedure

The *Rotate3D* procedure creates a matrix to rotate a vector. The parameters passed to this procedure are, first, an integer representing the axis about which the rotation is to take place (1 for the *x* axis, 2 for the *y* axis, or 3 for the *z* axis), and, second, a real number, which is the rotation angle in degrees. The procedure begins by zeroing the matrix. It sets the diagonal elements of the matrix not involved in the rotation to one. It then determines the indices for the matrix elements which perform the rotation. The cosine and sine of the rotation angle are next calculated. These two variables are then placed in the proper matrix elements (with the proper sign) to perform the rotation. The procedure returns the completed rotation matrix. The rotation matrix around the *x* axis is:

$$R = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & \sin\theta & 0 \\ 0 & -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (\text{Equation 1-10})$$

The rotation matrix around the y axis is:

$$R = \begin{bmatrix} \cos\theta & 0 & -\sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (\text{Equation 1-11})$$

The rotation matrix around the z axis is:

$$R = \begin{bmatrix} \cos\theta & \sin\theta & 0 & 0 \\ -\sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (\text{Equation 1-12})$$

The *Multiply3DMatrices* Procedure

The *Multiply3DMatrices* procedure performs the multiplication of two four-by-four, real-number matrices. The procedure uses three nested *For* loops to do the necessary element multiplications and additions needed to obtain each element of the new matrix.

The *PrepareMatrix* Procedure

The *PrepareMatrix* procedure creates a complete affine transformation matrix. The parameters passed to this procedure are the translation distances and scale factors in the x, y, and z directions, and the rotation angles around the x, y, and z axes. The procedure uses the *Scale3D* procedure to create the matrix for the rescaling-in-size operation. It then uses the *Rotate3D* procedure to generate the rotation matrices around each of the three axes, three times. Then it creates the translation matrix using the *Translate3D* procedure. Finally, it multiplies all of these matrices together to create the final affine transformation matrix, which is returned by the procedure.

The *PrepareInvMatrix* Procedure

The *PrepareInvMatrix* procedure begins the same way as the *PrepareMatrix* procedure (using the various procedures to create all parts of the transformation process). However, it multiplies these matrices together in the reverse order so the transformation is the inverse of the previous procedure.

The Transform Procedure

The *Transform* procedure performs the multiplication of a real three-dimensional vector by a real, four-by-four matrix. The returned result is a new, real, three-dimensional vector.

CHAPTER 2

The Graphics Interface Module

This chapter describes a number of graphics functions and procedures to paint our graphics to the display screen. These routines are included in a module called *Graph.Inc.*, listed in Figure 2-1. The various routines are described below. Refer to the listing for the source code as you read each description.

The *SetMode* Procedure

This procedure uses the ROM BIOS video services to set the computer to the desired display mode. The mode number passes to the procedure, which properly sets the color registers and initiates an interrupt causing BIOS to execute the mode change.

The *PreCalc* Procedure

This procedure initializes an array called *PreCalcY*. Each entry of this array consists of the display *x* resolution multiplied by the row number, which is the offset needed by the *Plot* procedure to properly locate the memory address to which it transfers information. Using this pre-initialized array reduces computation during plotting thereby speeding up the plotting procedure.

The *Plot* Procedure

This procedure plots a point to the screen in a designated color. The variables passed to the procedure are *x*, the horizontal position on the screen; *y*, the vertical position on the screen; and *Color* which is the palette number for the selected color. The procedure begins by checking whether *x* and *y* are within the limits set for the

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

display. If they are within limits, the memory offset for the display data is calculated. The display memory begins at address *A000:0000*. All that is necessary to find the correct pixel address for the 256 color display mode is to add the proper value of *x* and the proper value from the *PreCalcY* array. The *Color* value is then sent to this memory address.

{

Video Graphics Array Driver

ClearPalette	- clear palette register
SetPalette	- set palette register
InitPalette1	- 64 levels of gray, red, green and blue
InitPalette2	- 7 colors with 35 intensities each - use with Pixel
CyclePalette	- cycle through palette
Circle	- circle draw routine
Draw	- line draw routine
InitGraphics	- initialize graphics
WaitForKey	- wait for keypress
ExitGraphics	- sound and wait for keypress before exiting graphics
Title	- set up text screen colors

}

Var

Regs : Registers;

Procedure SetMode(Mode : Byte);

Begin

With Regs Do

Begin

AH := 0;

AL := Mode

End;

THE GRAPHICS INTERFACE MODULE

```
Intr($10, Regs)
End;
```

```
Const
```

```
MaxXRes = 320;
MaxYRes = 200;
MaxX    = (MaxXRes-1);
MaxY    = (MaxYRes-1);
```

```
Var
```

```
XRes, YRes : Integer;
PreCalcY   : Array[0..MaxY] Of Word;
```

```
Procedure PreCalc;
```

```
Var
```

```
  j : Word;
```

```
Begin
```

```
  For j := 0 To MaxY Do
    PreCalcY[j] := XRes * j
```

```
End;
```

```
Procedure Plot(x, y : Word; Color : Byte);
```

```
Var
```

```
  Offset : Word;
```

```
Begin
```

```
  If Not((x < 0) Or (y < 0) Or (x > MaxX) Or (y > MaxY)) Then
```

```
    Begin
```

```
      Offset := PreCalcY[y] + x;
```

```
      MEM[$A000:Offset] := Color
```

```
    End
```

```
  End;
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
Type
  RGB = Record
    Red, Grn, Blu : Byte
  End;
  PaletteRegister = Array[0..255] Of RGB;
```

```
Var
  Color : PaletteRegister;
```

```
Procedure ClearPalette(Var Color : PaletteRegister);
```

```
  Var
    i : Byte;

  Begin
    For i := 0 To 255 Do
      Begin
        Color[i].Red := 0;
        Color[i].Grn := 0;
        Color[i].Blu := 0
      End
    End;
End;
```

```
Procedure SetPalette(Hue : PaletteRegister);
```

```
  Begin
    With Regs Do
      Begin
        AX := $1012;
        BX := 0;
        CX := 256;
        ES := Seg(Hue);
        DX := OfS(Hue)
      End;
    Intr($10, Regs)
  End;
```

```
Procedure InitPalettel(Var Color : PaletteRegister);
```


THE GRAPHICS INTERFACE MODULE

```
Var  
  i : Byte;
```

```
Begin  
  For i := 0 To 63 Do    { Gray 0-63 }  
    Begin  
      Color[i].Red := i;  
      Color[i].Grn := i;  
      Color[i].Blu := i  
    End;  
  For i := 64 To 127 Do  { Red 0-63 }  
    Begin  
      Color[i].Red := i - 64;  
      Color[i].Grn := 0;  
      Color[i].Blu := 0  
    End;  
  For i := 128 To 191 Do { Green 0-63 }  
    Begin  
      Color[i].Red := 0;  
      Color[i].Grn := i - 128;  
      Color[i].Blu := 0  
    End;  
  For i := 192 To 255 Do { Blue 0-63 }  
    Begin  
      Color[i].Red := 0;  
      Color[i].Grn := 0;  
      Color[i].Blu := i - 192  
    End  
End;  
End;
```

```
Procedure InitPalette2(Var Color : PaletteRegister);
```

```
Var  
  i : Byte;
```

```
Begin  
  For i := 0 To 35 Do    { Blue }  
    Begin  
      Color[i].Red := 0;
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
        Color[i].Grn := 0;
        Color[i].Blu := Round((i+28) * 1.8)
    End;
For i := 36 To 71 Do { Green }
    Begin
        Color[i].Red := 0;
        Color[i].Grn := Round((i-8) * 1.8);
        Color[i].Blu := 0
    End;
For i := 72 To 107 Do { Cyan }
    Begin
        Color[i].Red := 0;
        Color[i].Grn := Round((i-44) * 1.8);
        Color[i].Blu := Round((i-44) * 1.8)
    End;
For i := 108 To 143 Do { Red }
    Begin
        Color[i].Red := Round((i-80) * 1.8);
        Color[i].Grn := 0;
        Color[i].Blu := 0
    End;
For i := 144 To 179 Do { Magenta }
    Begin
        Color[i].Red := Round((i-116) * 1.8);
        Color[i].Grn := 0;
        Color[i].Blu := Round((i-116) * 1.8)
    End;
For i := 180 To 215 Do { Brown }
    Begin
        Color[i].Red := Round((i-152) * 1.8);
        Color[i].Grn := Round((i-152) * 1.8);
        Color[i].Blu := 0
    End;
For i := 216 To 251 Do { Gray }
    Begin
        Color[i].Red := Round((i-188) * 1.8);
        Color[i].Grn := Round((i-188) * 1.8);
        Color[i].Blu := Round((i-188) * 1.8)
    End
End;
```


THE GRAPHICS INTERFACE MODULE

```
Procedure CyclePalette(Var Hue : PaletteRegister);
```

```
  Var
```

```
    i      : Byte;
```

```
    Tmp    : RGB;
```

```
  Begin
```

```
    Tmp := Hue[0];
```

```
    For i := 1 To 255 Do
```

```
      Hue[i-1] := Hue[i];
```

```
    Hue[255] := Tmp;
```

```
    SetPalette(Hue)
```

```
  End;
```

```
Procedure Swap(Var first, second : Integer);
```

```
  Var
```

```
    temp : Integer;
```

```
  Begin
```

```
    temp := first;
```

```
    first := second;
```

```
    second := temp
```

```
  End;
```

```
Procedure Circle(x, y, Radius : Word; Color : Byte);
```

```
  Var
```

```
    a, af, b, bf, target, r2 : Integer;
```

```
  Begin
```

```
    target := 0;
```

```
    a := radius;
```

```
    b := 0;
```

```
    r2 := Sqr(radius);
```

```
    While a >= b Do
```

```
      Begin
```

```
        b := Round(Sqrt(r2 - sqr(a)));
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
Swap( target, b );
While b < target Do
  Begin    { Inner loop for straight lines at
            cardinal points }
    af := (120 * a) Div 100; { aspect correction }
    bf := (120 * b) Div 100;
    Plot(x + af, y + b, color); { 8 points plotted
                                - symmetry }
    Plot(x + bf, y + a, color);
    Plot(x - af, y + b, color);
    Plot(x - bf, y + a, color);
    Plot(x - af, y - b, color);
    Plot(x - bf, y - a, color);
    Plot(x + af, y - b, color);
    Plot(x + bf, y - a, color);
    b := b + 1
  End;
  a := a - 1
End
End;
```

```
Procedure Draw( xx1, yy1, xx2, yy2 : Word; color : Byte );

Var
  LgDelta, ShDelta, Cycle, LgStep, ShStep, dtotal : Integer;

Begin
  LgDelta := xx2 - xx1; { delta and step calculations }
  ShDelta := yy2 - yy1;
  If LgDelta < 0 Then
    Begin
      LgDelta := -LgDelta;
      LgStep := -1
    End
  Else
    LgStep := 1;
  If ShDelta < 0 Then
    Begin
      ShDelta := -ShDelta;
```


THE GRAPHICS INTERFACE MODULE

```
        ShStep := -1
    End
Else
    ShStep := 1;
    If ShDelta < LgDelta Then
        Begin { X-axis longer => Cycle = LargeDelta / 2 and use
                steps as-is }

            Cycle := LgDelta SHR 1;
            While xx1 <> xx2 Do { While not at termination }
                Begin
                    Plot(xx1,yy1,color);
                    Cycle := Cycle + ShDelta;
                    If Cycle > LgDelta Then
                        Begin
                            Cycle := Cycle - LgDelta;
                            yy1 := yy1 + ShStep
                        End;
                        xx1 := xx1 + LgStep
                    End;
                    Plot(xx1,yy1,color)
                End
            End
        Else
            Begin { X-axis Shorter => Cycle = ShortDelta / 2 and
                    switch steps }

                Cycle := ShDelta SHR 1;
                Swap(LgDelta, ShDelta);
                Swap(LgStep, ShStep);
                While yy1 <> yy2 Do { While not at termination }
                    Begin
                        Plot(xx1,yy1,color);
                        Cycle := Cycle + ShDelta;
                        If Cycle > LgDelta Then
                            Begin
                                Cycle := Cycle - LgDelta;
                                xx1 := xx1 + ShStep
                            End;
                            yy1 := yy1 + LgStep
                        End;
                        Plot(xx1,yy1,color)
                    End
                End
            End
        End
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

End;

Procedure InitGraphics;

Begin

 XRes := MaxXRes;

 YRes := MaxYRes;

 PreCalc;

 SetMode(19);

 ClearPalette(Color);

 InitPalette2(Color);

 SetPalette(Color)

End;

Procedure WaitForKey;

Var

 ch, c : Char;

Begin

 Repeat

 Repeat

 ch := ReadKey

 Until (ch<>#0);

 c := UpCase(ch);

 Until (c > '')

End;

Procedure ExitGraphics;

Begin

 Sound(1000);

 Delay(500);

 NoSound;

 WaitForKey;

 SetMode(3)

End;

THE GRAPHICS INTERFACE MODULE

Procedure Title;

Begin

TextColor(Yellow);

TextBackground(Blue);

ClrScr

End;

{

Three-Dimensional Plotting Routines

InitPlotting - rotation and tilt angles
InitPerspective - observer location and distances
MapCoordinates - maps 3D space onto the 2D screen
CartesianPlot3D - plot a cartesian system point
CylindricalPlot3D - plot a cylindrical system point
SphericalPlot3D - plot a spherical system point
DrawLine3D - plots a line from 3D coordinates

}

Var

CentreX, CentreY : Integer;

Angl, Tilt : Integer;

CosA, SinA : Real;

CosB, SinB : Real;

CosACosB, SinASinB : Real;

CosASinB, SinACosB : Real;

Procedure InitPlotting(Ang, Tlt : Integer);

Begin

CentreX := MaxX Div 2;

CentreY := MaxY Div 2;

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
    Angl := Ang;
    Tilt := Tlt;
    WriteLn('View Direction is ', Angl:4, '° around the z-Axis
            and');
    WriteLn('          ', Tilt:4, '° off of the z-Axis');
    CosA := CosD( Angl );
    SinA := SinD( Angl );
    CosB := CosD( Tilt );
    SinB := SinD( Tilt );
    CosACosB := CosA * CosB;
    SinASinB := SinA * SinB;
    CosASinB := CosA * SinB;
    SinACosB := SinA * CosB
End;
```

```
Var
    PerspectivePlot : Boolean;
    Mx, My, Mz, ds : Real;
```

```
Procedure InitPerspective(Perspective : Boolean; x, y, z, m :
Real);
```

```
    Begin
        PerspectivePlot := Perspective;
        Mx := x;
        My := y;
        Mz := z;
        ds := m
    End;
```

```
Procedure MapCoordinates(X, Y, Z : Real; Var Xp, Yp : Integer);
```

```
    Var
        Xt, Yt, Zt : Real;
```

```
    Begin
        Xt := (Mx + X * CosA - Y * SinA);
```


THE GRAPHICS INTERFACE MODULE

```
Yt := (My + X * SinASinB + Y * CosASinB + Z * CosB);
If PerspectivePlot Then
  Begin
    Zt := Mz + X * SinACosB + Y * CosACosB - Z * SinB;
    Xp := CentreX + Round( ds * Xt / Zt );
    Yp := CentreY - Round( ds * Yt / Zt )
  End
Else
  Begin
    Xp := CentreX + Round( Xt );
    Yp := CentreY - Round( Yt )
  End
End;
```

```
Procedure CartesianPlot3D(X, Y, Z : Real; Color : Byte);
```

```
  Var
    Xp, Yp : Integer;

  Begin
    MapCoordinates(X, Y, Z, Xp, Yp);
    Plot( Xp, Yp, Color )
  End;
```

```
Procedure CylindricalPlot3D(Rho, Theta, Z : Real; Color : Byte);
```

```
  Var
    X, Y : Real;

  Begin
    Theta := Radians( Theta );
    X := Rho * Cos( Theta );
    Y := Rho * Sin( Theta );
    CartesianPlot3D( X, Y, Z, Color )
  End;
```

```
Procedure SphericalPlot3D(R, Theta, Phi : Real; Color : Byte);
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
Var
  X, Y, Z : Real;

Begin
  Theta := Radians( Theta );
  Phi   := Radians( Phi );
  X := R * Sin( Theta ) * Cos( Phi );
  Y := R * Sin( Theta ) * Sin( Phi );
  Z := R * Cos( Theta );
  CartesianPlot3D( X, Y, Z, Color )
End;
```

Procedure DrawLine3D(Pnt1, Pnt2 : TDA; Color : Byte);

```
Var
  Xp1, Yp1 : Integer;
  Xp2, Yp2 : Integer;
  x1, y1, z1 : Real;
  x2, y2, z2 : Real;

Begin
  UnVec(Pnt1, x1, y1, z1);
  UnVec(Pnt2, x2, y2, z2);
  MapCoordinates(x1, y1, z1, Xp1, Yp1);
  MapCoordinates(x2, y2, z2, Xp2, Yp2);
  Draw(Xp1, Yp1, Xp2, Yp2, Color)
End;
```

```
{ Color 1 - Blue      }
{ 2 - Green           }
{ 3 - Cyan            }
{ 4 - Red              }
{ 5 - Magenta          }
{ 6 - Brown/Yellow    }
{ 7 - Gray Scale      }
```

```
{ Intensity levels (0..35) for each color }
```


THE GRAPHICS INTERFACE MODULE

Const

MaxCol = 7;

MaxInten = 35;

{

PutPixel - plots a pixel to the screen

}

Procedure PutPixel(x, y : Integer; Color, Intensity : Byte);

Var

Col : Byte;

Begin

If Intensity > MaxInten Then Intensity = MaxInten;

Col := (MaxInten + 1) * (Color-1) + Intensity;

Plot(x, y, Col)

End;

{

GetPixel - gets a pixel from the screen

}

Function GetPixel(x, y : Integer) : Integer;

Begin

Regs.AX := 3328;

Regs.DX := y;

Regs.CX := x;

Intr(\$10, Regs);

GetPixel := Regs.AX And 255

End;

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

Setup of Coordinate Axes and Color Palette

```
PutAxisAndPalette - toggle for Axis and Palette
AxisAndPalette - places Axis and Color Palette on screen
```

}

```
Var
    DrawAxisAndPalette : Boolean;
```

```
Procedure PutAxisAndPalette(PlaceOnScreen : Boolean);
```

```
Begin
  If PlaceOnScreen Then
    DrawAxisAndPalette := True
  Else
    DrawAxisAndPalette := False
End;
```

```
Procedure AxisAndPalette;
```

```
Procedure DisplayAxis;
```

```
Var
    x, y, z : Integer;
```

```
Begin
  For x := -100 To 100 Do CartesianPlot3D(x, 0, 0, 35);
                                { Blue }
  CartesianPlot3D(100, 0, 0, 251);
                                { White }
  For y := -100 To 100 Do CartesianPlot3D(0, y, 0, 71);
                                { Green }
  CartesianPlot3D(0, 100, 0, 251);
                                { White }
  For z := -100 To 100 Do CartesianPlot3D(0, 0, z, 107);
                                { Cyan }
```


THE GRAPHICS INTERFACE MODULE

```
CartesianPlot3D(0, 0, 100, 251) { White }
End;

Procedure DisplayPalette;

Var
  X, Y : Integer;
  Color : Byte;
  Intensity : Byte;

Begin
  For Color := 1 To MaxCol Do
    For Intensity := 0 To MaxInten Do
      For X := 0 To 3 Do
        For Y := 0 To 3 Do
          PutPixel(X + 5*Color, 190 - Y - 5*Intensity,
                  Color, Intensity)
        End;
      End;
    End;
  End;

Begin
  If DrawAxisAndPalette Then
    Begin
      DisplayAxis;
      DisplayPalette
    End
  End;
End;
```

Figure 2-1. *Graph.Inc.* Module

The *ClearPalette* Procedure

The array *PaletteRegister* contains the red, green, and blue components that define the color for each of the 256 color registers. This procedure sets all of these values to zero to clear the array.

The *SetPalette* Procedure

This procedure uses the ROM BIOS video services to set the 256 color registers to the 256 colors defined by the array *PaletteRegister*. The procedure sets the mi-

coprocessor registers to call the proper ROM BIOS function and indicate that it is to transfer information to the VGA card from the address that marks the beginning of the *PaletteRegister* array. The video services interrupt performs this operation.

The *InitPalette1* Procedure

This procedure sets up the color designations in the *PaletteRegister* array. The initial color selections chosen by IBM are compatible with the EGA, but otherwise are rather useless in the 256-color mode. They begin by assigning the first 64 colors to the 64 colors which can be produced by the EGA. Next are 64 shades of gray. The origin of the shades assigned to the remaining color registers are rather obscure. In most cases, you can replace these default colors with a set of colors that are more adapted to the types of displays you'll produce. Using this procedure, divide the color registers into four basic groups; gray, red, green, and blue. In the gray group, assign each of the three primary colors (red, green, and blue) to the same intensity value for each shade, with the shades varying from 0 to 63. For the red group, assign red to intensities from 0 to 63 and leave the other colors at zero. For the green group, assign green to intensities from 0 to 63 and leave the other colors at zero. For the blue group, assign blue to intensities from 0 to 63 and leave the other colors at zero.

The *InitPalette2* Procedure

This procedure sets up a different set of color designations in the *PaletteRegister* array (which may be more appropriate for particular displays we generate). Using this procedure, divide the color registers into seven basic groups; blue, green, cyan, red, magenta, brown, and gray. For the red group assign intensities of 0 to 63 to red and leave the other colors at zero. The cyan group has 35 levels of intensity from 0 to 63 with equal values of green and blue. The magenta is similar with equal values of red and blue. The brown group is similar with equal values of red and green. The gray group is similar with equal values of all three colors. For the green group, assign green to 35 levels of intensity from 0 to 63 and leave the other colors at zero. For the blue group, assign blue to intensities from 0 to 63 and leave the other colors at zero.

The Cycle Palette Procedure

This procedure cycles all of the colors in the *PaletteRegister* array. It saves the first set of color values to a temporary storage variable. It initiates a *For* loop which moves all of the remaining 255 sets of color values to the next lower member of the array. When the loop is complete, the beginning value is transferred from the temporary variable to the highest member of the array. Note that this cycles the color values in the array, but has no effect upon the actual colors of the display. To change these colors, the *SetPalette* procedure must be called to transfer the array information to the VGA color registers.

The Swap Procedure

This procedure simply swaps two integers. In other words, if *first* contains the value 3 and *second* contains the value 4, then after *Swap* is called, *first* will contain 4 and *second* will contain 3. This is an operation that is often needed for our graphics programs, so you'll have made a separate procedure for it.

The Circle Procedure

This procedure is used to draw a circle on the screen (in the 320-by-200-pixel-by-256-color mode) when the coordinates of the circle center, its radius color are specified. Note that there is a symmetry in the circle that permits us to specify offsets from the center of the circle in both the *x* and *y* directions and by assigning the proper signs to these offsets, draw eight different points on the circle. By performing these eight plots for each offset computation, you have drawn the entire circle, minimizing the required computations. If the pixels are square, you'll need only two offsets. Since they are not, you'll need to multiply the offsets for the *x* coordinate by a factor (1.2) to correct the aspect of the circle. Now, take a look at how the computations are performed. First, one offset is assigned as the radius of the circle. The second and a target value, are set to zero. Enter a *While* loop which continues until the second offset is less than the first (indicating that one-eighth of the circle is complete). Within this loop, the procedure first computes the value of the second offset, using the equation of a circle and swaps it with the target value. A second *While* loop is entered, which iterates until the value of the second offset is greater than or equal to

the target value. For each iteration, the x offsets are computed (with factor to correct the aspect ratio) and the eight symmetrical points are then plotted. The second offset increments by one. (The reason for this loop is to fill in sections of the circle where the second offset changes so little that there is no change in terms of pixel location for that offset.) When this loop completes, decrement the first offset by one pixel and perform another iteration of the first loop. The result is the most accurate circle allowable for the screen resolution.

The *Draw* Procedure

This procedure draws a line from one specified point to another, using a designated color. The procedure begins by calculating *LgDelta* (the distance between the beginning and ending points in the x direction), and *ShDelta* (the distance between the beginning and end points in the y direction). Associated with each of these are step parameters (*LgStep* and *ShStep*). If either of the distance parameters is negative, its sign is changed and the corresponding step parameter is -1 ; otherwise it is $+1$. The rest of the procedure is determined by an *If* statement which directs one course of action if the y line distance is shorter than the x line distance. Otherwise, it will take another course of action.

Let's start by looking at the first situation. First compute a parameter *Cycle* which is half of the x line distance (the longer distance). Next enter a *While* loop which iterates as long as the x pixel coordinate (*xx1*) is less than the ending x line coordinate. Initially, this coordinate is the beginning line x coordinate. First, a pixel is plotted at the x and y coordinates in the designated color. Next *Cycle* is increased by the shorter distance. If this makes it larger than the longer distance, it is reduced by the longer distance and the y coordinate is changed by adding the *ShStep* parameter (which is $+1$ or -1). Then the x coordinate is changed by the *LgStep* parameter (also $+1$ or -1). The loop reiterates, by plotting the new pixel location. When the loop terminates, the coordinates are positioned at the end of the line, but the final point has not yet been plotted. The first thing that happens after loop termination is to plot this final point.

Now let's look at what happens when the y line distance is longer than the x line distance. First, the *Cycle* parameter is computed to be one-half of *ShDelta* instead of one-half of *LgDelta*. Next the values of *LgDelta* and *ShDelta* are swapped and the

values of *LgStep* and *ShStep* are swapped. The rest of the procedure is like the previous loop described except that the termination of the loop depends on the value of the *y* pixel parameter instead of the *x*. Also, the order of changing the *x* and *y* pixel coordinates is reversed. (This procedure is an implementation of Bresenham's algorithm, which is generally accepted as the best method for drawing lines on a screen.)

The *InitGraphics* Procedure

This procedure initializes the screen for graphics. It begins by setting the *XRes* and *YRes* parameters to their maximum values. It calls *SetMode* to set the graphics mode to 19, which is the 320-by-200-pixel-by-256-color mode. It calls *PreCalc* to fill the *PreCalcY* array, *ClearPalette* to clear the *Color* array, followed by *InitPalette2* to set up the desired colors in the *Color* array. Finally, it calls *SetPalette* to store the colors designated in the *Color* array into the 256 color registers of the VGA board.

The *WaitForKey* Procedure

This procedure stops all program action except searching for a key to be pressed at the keyboard. The procedure terminates when a key having an ASCII value greater than a space is entered. This includes all alphanumerics.

The *ExitGraphics* Procedure

This procedure is usually called at the end of generating a display picture. It begins with a half-second tone, to alert the user that the display is complete. It calls *WaitForKey*, awaiting a key stroke on the keyboard. When the keystroke is entered, it resets the display to the text mode.

The *Title* Procedure

This procedure prepares the screen for writing information in the text mode. It sets the color of the text to yellow and the background to blue. The screen clears, leaving the blue background.

Three-Dimensional Plotting Routines

The remaining procedures listed in the graphics module are used for generating three-dimensional pictures. They include procedures for initializing the rotation and tilt angles, setting up the observer location and distances, mapping the three-dimensional space onto a two-dimensional screen, plotting in Cartesian, cylindrical, or spherical coordinates, and drawing a line when given three-dimensional coordinates. These procedures are described in the following sections.

The *InitPlotting* Procedure

The *InitPlotting* procedure initializes variables used in three-dimensional plotting, and are dependent upon the rotation and tilt angles. These angles are passed to the procedure in integer form. The first initialized variables are *CentreX* and *CentreY* which are the coordinates of a point at the center of the screen. Next, the parameter *Angl* is set to the rotation angle passed to the function. The parameter *Tilt* is set to the tilt angle passed to the procedure. The procedure writes to the screen something like this: *View Direction is 45° around the z-Axis and 62° off of the z-Axis*, where the first angle will actually be the viewer rotation angle that was selected and the second angle will be the viewer tilt-angle selected. Next, the parameters *CosA*, *SinA*, *CosB*, and *SinB* are initialized, where *CosA* and *SinA* are the cosine and sine, respectively, of the rotation angle and *CosB* and *SinB* are the cosine and sine, respectively, of the tilt angle. Next, the parameters *CosACosB*, *SinASinB*, *CosASinB*, and *SinACosB*, where each parameter is the product of two trigonometric functions, are indicated by name.

The *InitPerspective* Procedure

The *InitPerspective* procedure is passed a Boolean flag indicating that perspective plotting is to take place, together with the *x*, *y*, and *z* coordinates of the observer position in space and a parameter representing the observer distance from the display screen. The flag is stored in the variable *PerspectivePlot*. The observer position coordinates are stored in *Mx*, *My*, and *Mz* respectively. The distance is stored in the parameter *ds*.

The *MapCoordinates* Procedure

The coordinates of a point in three-dimensional space (X, Y, Z) pass to this procedure as real numbers. These are in a coordinate system whose origin is at the center of the display screen. First, the proper mathematical expressions are solved to project this point onto the two-dimensional screen, yielding the two-dimensional coordinates X_t and Y_t . These expressions are:

$$x_t = m_x + x \cos \theta - y \sin \theta \quad (\text{Equation 2-1})$$

and

$$y_t = m_y + x \sin \theta \sin \phi + y \cos \theta \sin \phi + z \cos \phi \quad (\text{Equation 2-2})$$

where x_t and y_t are the two-dimensional coordinates not yet converted to display coordinates. The angles and x, y, z coordinates can be seen in Figures 2-2, 2-3, and 2-4. The parameters m_x , m_y , and m_z represent the viewer's position and are used when perspective is necessary. If there is no perspective, these parameters are zero. Next, the procedure checks whether a perspective plot is necessary. If so, it computes a z coordinate as follows:

$$z_t = m_z + x \sin \theta \cos \phi + y \cos \theta \cos \phi - z \sin \phi \quad (\text{Equation 2-3})$$

The procedure divides the x and y coordinates by this z coordinate, multiplies them by the viewer distance from the screen, and adds in the center-of-screen coordinates to convert the pair to integers, X_p and Y_p . They are the column and row representation of the pixel plotted to the screen. If a perspective plot is not necessary, the originally projected coordinate results modify to the column and row (X_p and Y_p) by adding the center-of-screen coordinates. In either case, the results are returned by the procedure.

The *CartesianPlot3D* Procedure

This plots a point to the screen that passes to the procedure as the x , y , and z coordinates of the point in three-dimensional space, plus the designated color for the

point. Figure 2-2 shows a point plotted in Cartesian coordinates. The procedure calls *MapCoordinates* to convert from the three-dimensional coordinates to the display screen coordinates. It plots a pixel at these coordinates in the designated color.

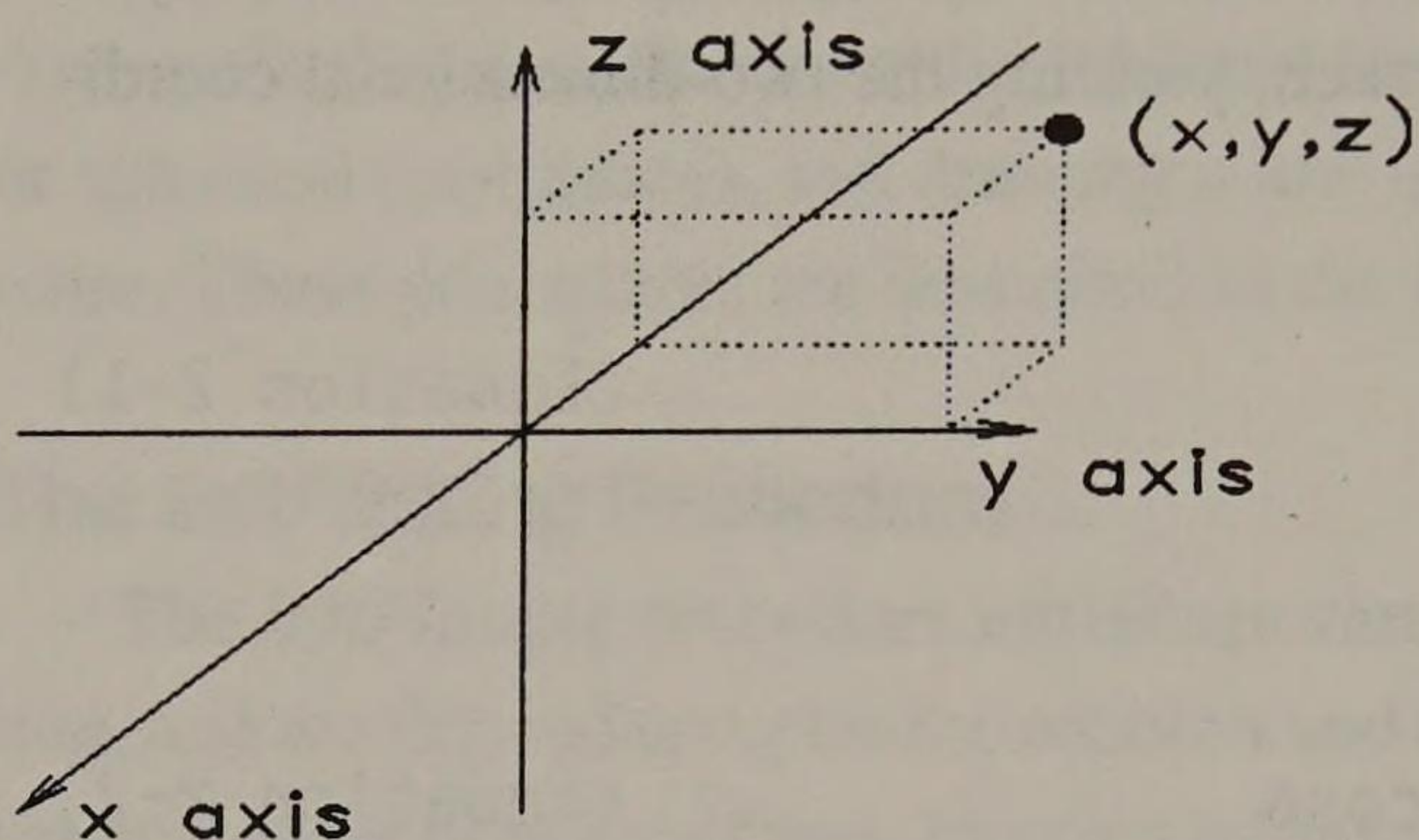


Figure 2-2. Cartesian Coordinates

The *CylindricalPlot3D* Procedure

This plots a point that passes to the procedure in the form of cylindrical coordinates *Rho* (radius), *Theta* (angle of radius line in the *xy* plane), *Z* (height), and a designated color. A point plotted in cylindrical coordinates is shown in Figure 2-3. First, the angle *Theta* converts from degrees to radians and the trigonometric relationships are solved to find the three-dimensional coordinates *X* and *Y*. The procedure *CartesianPlot3d* plots a pixel on the two dimensional screen using the coordinates *X*, *Y*, and *Z* and the designated color.

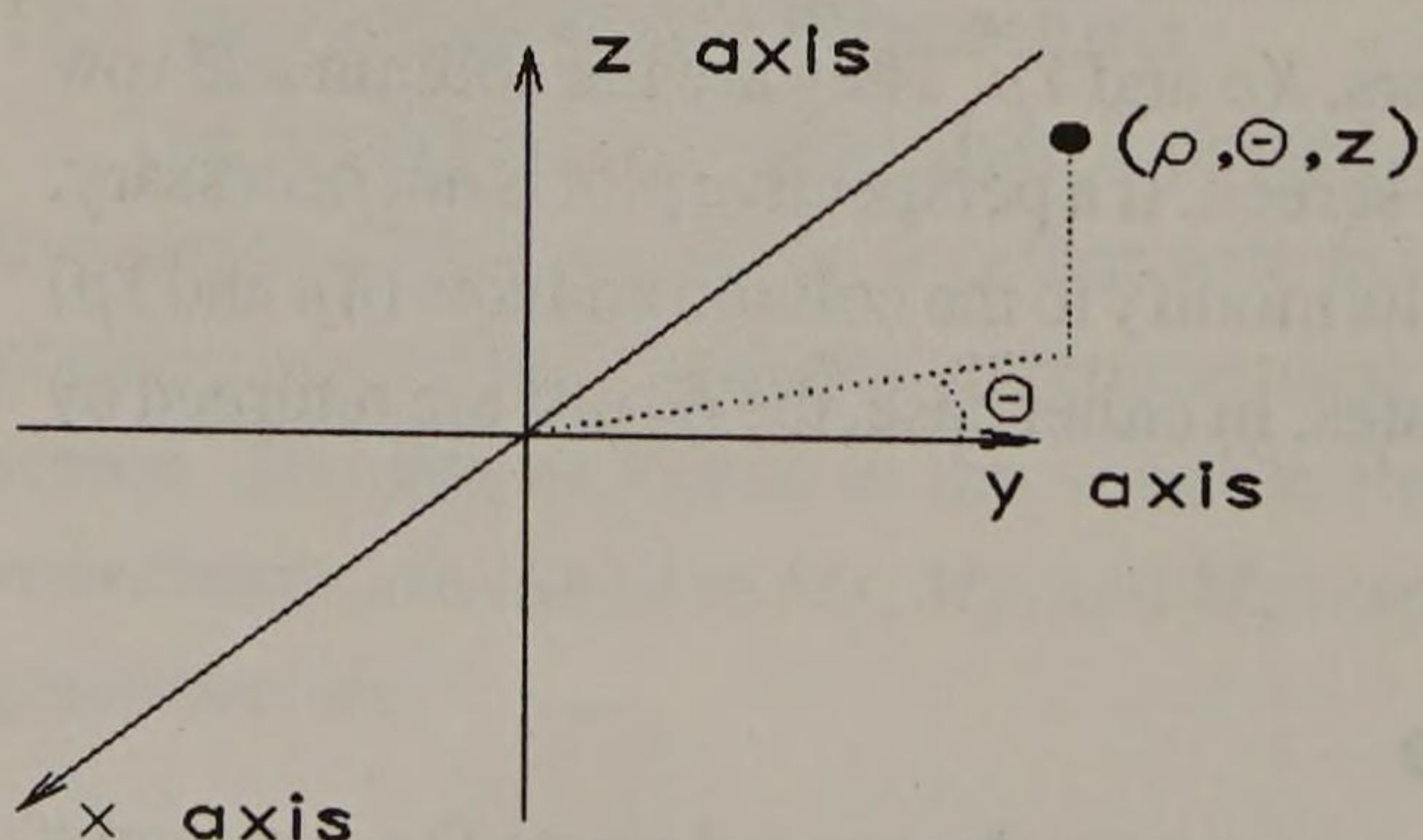


Figure 2-3. Cylindrical Coordinates

The *SphericalPlot3D* Procedure

This procedure is used to plot a point to the screen that is passed to the procedure in the form of spherical coordinates r (radius), Θ (angle of radius line in the xy plane), and Φ (angle of the radius line in the xz plane), and a designated color. A point plotted in spherical coordinates is shown in Figure 2-4. First, the angles Θ and Φ are converted from degrees to radians and the trigonometric relationships are solved to find the three-dimensional coordinates X , Y , and Z . The procedure *CartesianPlot3d* is called to plot a pixel on the two-dimensional screen using coordinates X , Y , and Z and the designated color.

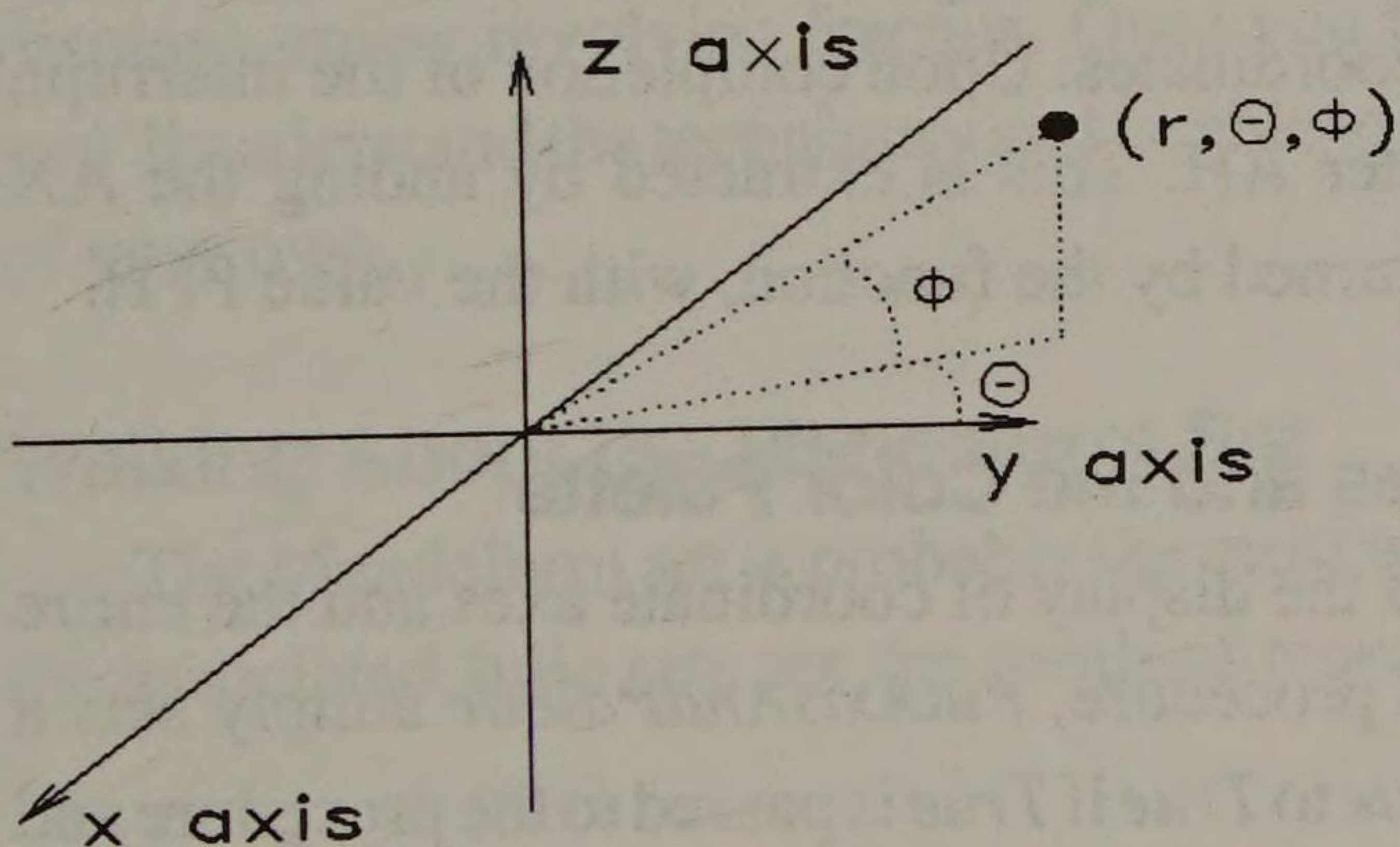


Figure 2-4. Spherical Coordinates

The *DrawLine3D* Procedure

Two three-dimensional vectors representing the beginning and end of the line, and a designated color, pass to this procedure. It first uses the procedure *UnVec* to convert each vector into three separate coordinates. It calls *MapCoordinates* twice. The first call converts three-dimensional, beginning coordinates of the line to two-dimensional display screen coordinates. The second call converts three-dimensional, ending coordinates of the line to two-dimensional display screen coordinates. Finally, the procedure calls *Draw* to draw a line on the screen from the beginning coordinates to the ending coordinates in the designated color.

The *PutPixel* Procedure

The *PutPixel* Procedure plots a pixel on the screen when the shade of color is given as a basic color number and intensity level. The procedure first checks that the intensity does not exceed the maximum level. It computes the color register number (from 1 to 255) of the desired color shade, and *Plot* paints the selected color pixel on the screen at the specified coordinates.

The *GetPixel* Function

This function uses the ROM BIOS video services to obtain a pixel color at a set of coordinates on the screen. The microprocessor registers initiate the video service interrupt and pass the proper screen coordinates. Upon completion of the interrupt, the pixel color is contained in register AH. This is extracted by anding the AX register, which contains the value returned by the function, with the value FFH.

Displaying the Coordinate Axes and the Color Palette

This group of procedures can add the display of coordinate axes and the entire color palette to any screen. The first procedure, *PutAxisAndPalette* simply sets a Boolean variable: *DrawAxisAndPalette* to *True* if *True* is passed to the procedure and to *False*, if *False* is passed to the procedure. The next procedure, *AxisAndPalette*, checks the above Boolean variable. If it is *True*, *DisplayAxis* draws the coordinate axes, and *DisplayPalette* creates the display of palette colors.

The procedure *DisplayAxis* uses *CartesianPlot3D* to convert points from -100 to +100 along the three coordinate axes to their two-dimensional equivalents and displays them on the screen as blue, green, and cyan lines for x, y, and z, respectively.

The procedure *DisplayPalette* uses nested *For* loops to generate 3-by-3 pixel blocks of each color available in the palette. These are located at the left hand side of the screen.

CHAPTER 3

How to Use the Modules

Now that you understand the use of the Mathematics and Graphics Interface modules, it is time to create some graphics displays using these new tools. In the following paragraphs, graphics programs are described for you to create interesting displays, many involving fractals. Once you become familiar with these displays, you'll understand the techniques and be able to generate new and interesting displays of your own.

Walking About the Mandelbrot Set

The Mandelbrot set is probably the most well-known of all fractal curves. It and the associated Julia sets are the result of repeated iteration of the equation:

$$z_n = z_{n-1}^2 + c$$

(Equation 3-1)

where z and c are complex numbers. To generate Mandelbrot sets, let the display plane represent a range of values for c . Each point on the plane corresponds to some pair of values for x (the real part of c) and y (the imaginary part of c). Starting with z_0 equal to zero, iterate the equation and plot a color to the pixel that represents the behavior of the iterated equation. To generate Julia sets, select some fixed value for c and let the display plane represent a range of starting values for z_0 . It is often said that the Mandelbrot set is a map of the various Julia sets. This is true because the Mandelbrot set maps the values of c , for each of which there is a unique Julia set, and because interesting features of the Mandelbrot set (such as cusps of the curve) usually correspond to interesting Julia sets.

To demonstrate the relationship, the program *MSetJset.Pas* first generates a Mandelbrot set on the left-hand side of the display and lets the user move a cursor around this display. At each cursor location, an inverse technique is used to generate

the corresponding Julia set on the right-hand side of the display. The Julia display continues to generate points (improving the detail of the rendition) until the user moves the cursor (whereupon the right-hand side of the display is blanked and generation of the new Julia set begins), or until the user hits the *Esc* key, which terminates the program. Figure 3-1 is a listing of the *MSetJSet.Pas* program.

The main program, at the end of the listing, simply calls five procedures. The first of these, *InitGraphics*, is described in the previous chapter. It sets the desired 320-by-200-pixel-by-256-color graphics mode. The next procedure, *InitCursor*, zeroes the members of *ColrArray*, which later stores cursor information. The next procedure is *CalcMSet*. This is the procedure that paints the Mandelbrot set on the screen. It uses the *Draw* procedure to draw a border for the Mandelbrot set around the left half of the screen and a border for the Julia set around the right half of the screen. It determines the limit of iterations, *L1* and *L2*, for which displayed colors are to change. Next, it determines the quantities by which *x* and *y* (the parameters of the Mandelbrot equation constant) increase for each pixel-size increment on the display.

The procedure enters a pair of nested *For* loops. The first iterates through values of *y*, which designate the pixel row on the screen, from the edge to the center of the screen. For each one, it determines the imaginary part of the value of the iterated equation constant, *cy*. The next loop iterates through the values of *x*, which designates the pixel column on the screen. First, *cx*, the value of the real part of the Mandelbrot constant is determined. Then the function *Iterate* iterates the equation. This function first sets variables *x* (the real part) and *y* (the imaginary part) to the corresponding constant values and finds their squares. Next, it begins a *While* loop which repeats until either the maximum number of iterations is reached or the value of the sum of the squares is greater than 4. Mathematically, the latter condition shows that the result of the equation is diverging, so it is time to quit. For each repetition of the *While* loop, the Mandelbrot equation is solved to find the next set of values and the number of iterations is incremented. When the loop terminates, the number of iterations is returned by the function. The *CalcMSet* procedure then determines the color, depending upon the number of iterations, and paints this color on the screen by calling the *Pix* procedure. (This procedure takes advantage of the symmetry of the Mandelbrot set to plot two pixels, one above and one below the center of the screen, using the *Plot* function, described in the previous chapter.)

HOW TO USE THE MODULES

```
{  
    MSet JSet.pas = Walkabout on the Mandelbrot Set-Display  
                    the Julia Set  
}
```

Uses

Dos, Crt;

{\$I Math.Inc}

{\$I Graph.Inc}

Const

XMin = -2.20;

XMax = 0.60;

YMin = -1.20;

YMax = 1.20;

MaxIter = 30;

Res = 160;

Procedure CalcMSet;

Function Iterate(cx, cy : Real) : Integer;

Var

Iters : Integer;

x, y : Real;

x2, y2 : Real;

temp : Real;

Begin

x := cx;

x2 := Sqr(x);

y := cy;

y2 := Sqr(y);

Iters := 0;

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
While (Iters < MaxIter) And (x2+y2 < 4) Do
    { while still converging }
```

```
Begin
```

```
    temp := cx + x2 - y2;
```

```
    {  $z = z^2 + c$  where  $z = x + yi$  }
```

```
    y := cy + 2 * x * y;
```

```
    y2 := Sqr(y);
```

```
    x := temp;
```

```
    x2 := Sqr(x);
```

```
    Iters := Succ(Iters)
```

```
End;
```

```
Iterate := Iters
```

```
End;
```

```
Procedure Pix(x, y, col : Integer);
```

```
Var
```

```
    a : Integer;
```

```
Begin
```

```
    Plot(x, y, col);
```

```
    Plot(x, Res-y-1, col);
```

```
End;
```

```
Var
```

```
    ix, iy : Integer;
```

```
    Iters : Integer;
```

```
    cx, cy : Real;
```

```
    dx, dy : Real;
```

```
    L1, L2 : Integer;
```

```
Begin
```

```
    Draw(0, 0, Res-1, 0, 35); { border Mandelbrot }
```

```
    Draw(Res-1, 0, Res-1, Res-1, 35);
```

```
    Draw(Res-1, Res, 0, Res, 35);
```

```
    Draw(0, Res, 0, 0, 35);
```

```
    Draw(Res, 0, 2*Res-1, 0, 35); {border Julia}
```

```
    Draw(2*Res-1, 0, 2*Res-1, Res-1, 35);
```

```
    Draw(2*Res-1, Res, Res, Res, 35);
```


HOW TO USE THE MODULES

```
{limits for color calc}
L1 := (MaxIter - (MaxIter * 2) Div 3);
L2 := (MaxIter - (MaxIter * 5) Div 6);

dx := (XMax - XMin) / (Res - 1);
dy := (YMax - YMin) / (Res - 1);

For iy := 2 To ((Res-1) Div 2) Do
  Begin
    cy := YMin + iy * dy;
    For ix := 2 To Res-3 Do
      Begin
        cx := XMin + ix * dx;
        Iters := Iterate(cx, cy);
        If (Iters = MaxIter) Then
          {color based on number of iterations }
          Pix(ix, iy, 143) { light red }
        Else
          If (Iters > L1) Then
            Pix(ix, iy, 215) { light yellow }
          Else
            If (Iters > L2) Then
              Pix(ix, iy, 35) { light blue }
            End
          End
        End
      End
    End
  End;

Procedure CalcJSet(cx, cy : Real);

Var
  xp, yp : Integer;
  dx, dy : Real;
  r      : Real;
  Theta  : Real;
  x, y   : Real;

Begin
  x := 0;
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
y := 0;
Repeat { calculate Julia Set by iterating inverse
                                         equation }
  dx := x - cx;
  dy := y - cy;
  If (dx > 0) Then
    Theta := ArcTan(dy / dx) * 0.5
  Else
    If (dx < 0) Then
      Theta := (Pi + ArcTan(dy / dx)) * 0.5
    Else
      Theta := Pi * 0.25;
  r := Sqrt( Sqrt(Sqr(dx) + Sqr(dy)) );
  If (Random < 0.5) Then r := -r;
  x := r * Cos(Theta);
  y := r * Sin(Theta);
  xp := Res Div 2 + Round(x * 36) + Res;
  yp := Res Div 2 + Round(y * 36);
  Plot(xp, yp, 143) { light red }
Until KeyPressed
End;
```

```
Var
  ColrArray : Array[1..9] Of Byte;
  xx        : Byte;
```

```
Procedure InitCursor;
```

```
  Begin
    For xx := 1 To 9 Do
      ColrArray[xx] := 0
  End;
```

```
Procedure PutCursor(x, y : Integer; Var ox, oy : Integer);
```

```
  Begin
    For xx := 1 To 5 Do { restore the screen }
```


HOW TO USE THE MODULES

```
    Plot(ox+xx-3, oy, ColrArray[xx]);
For xx := 6 To 7 Do
    Plot(ox, oy+xx-8, ColrArray[xx]);
For xx := 8 To 9 Do
    Plot(ox, oy+xx-7, ColrArray[xx]);

For xx := 1 To 5 Do { save the screen }
    ColrArray[xx] := GetPixel(x+xx-3, y);
For xx := 6 To 7 Do
    ColrArray[xx] := GetPixel(x, y+xx-8);
For xx := 8 To 9 Do
    ColrArray[xx] := GetPixel(x, y+xx-7);

Draw(x-2, y, x+2, y, 71); { draw green cursor }
Draw(x, y-2, x, y+2, 71);

ox := x;
oy := y
End;
```

Procedure BlockClearHalfScreen;

```
Var
    t, a, Offset : Word;

Begin
    a := 320;
    For t := 1 To Res-1 Do
        Begin
            For Offset := a To a+Res-3 Do
                MEM[$A000:Offset+Res] := 0;
            a := a + 320
        End
    End;
End;
```

```
Var
    cx, cy : Real;
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

Procedure ViewStats;

Var

Dir : Char;

Begin

ExitGraphics;

Title;

WriteLn('Mandelbrot Set :');

WriteLn(' XMin = ', XMin:4:4);

WriteLn(' XMax = ', XMax:4:4);

WriteLn(' YMin = ', YMin:4:4);

WriteLn(' YMax = ', YMax:4:4);

WriteLn;

WriteLn('Julia Set :');

WriteLn(' cx = ', cx:4:4);

WriteLn(' cy = ', cy:4:4);

Dir := ReadKey;

InitGraphics;

InitCursor;

CalcMSet

End;

Procedure WalkAbout;

Var

oCursX, oCursY : Integer;

CursX, CursY : Integer;

Dir : Char;

dx, dy : Real;

Begin

dx := (XMax - XMin) / (Res - 1);

dy := (YMax - YMin) / (Res - 1);

oCursX := 3;

oCursY := 3;

CursX := oCursX;

HOW TO USE THE MODULES

```
CursY := oCursY;
PutCursor(CursX, CursY, oCursX, oCursY);

Repeat
  Dir := ReadKey;
  If (Dir = #0) Then Dir := ReadKey;
  Case Dir Of
    ' ' : ViewStats;
    'K' : CursX := CursX - 1;      { left arrow }
    'M' : CursX := CursX + 1;      { right arrow }
    'H' : CursY := CursY - 1;      { up arrow }
    'P' : CursY := CursY + 1;      { down arrow }
    'G' : Begin                    { Home }
      CursX := CursX - 1;
      CursY := CursY - 1
    End;
    'I' : Begin                    { PgUp }
      CursX := CursX + 1;
      CursY := CursY - 1
    End;
    'O' : Begin                    { End }
      CursX := CursX - 1;
      CursY := CursY + 1
    End;
    'Q' : Begin                    { PgDn }
      CursX := CursX + 1;
      CursY := CursY + 1
    End
  End;
  PutCursor(CursX, CursY, oCursX, oCursY);

  cx := XMin + dx * CursX;  { calculate c }
  cy := YMin + dy * CursY;

  BlockClearHalfScreen;      { clear the old Julia set }
  CalcJSet(cx, cy)           { calculate new Julia Set }

Until (Dir = #27)
```



```

    End;

Begin
    InitGraphics;

    InitCursor;

    CalcMSet;

    WalkAbout;

    ExitGraphics
End.

```

Figure 3-1. Listing of Program to Walk About the Mandelbrot Set

Once the Mandelbrot set has been plotted, the main program calls the procedure *WalkAbout*, which allows “walking about” the Mandelbrot set with the cursor and plotting the selected Julia set. The procedure initializes the parameters that store the cursor location. Next it calls *PutCursor*, which draws the cursor on the screen. *PutCursor* first takes the stored information from the array *ColrArray* and uses it to restore the screen by redrawing the cursor image (a cross 5-by-5 pixels) in original colors at its current location. It fills *ColrArray* with the necessary information from the new location of the cursor and draws the cursor at its new location. It stores the new cursor location for restoring the scene data the next time the procedure is run.

Returning to the *WalkAbout* procedure, you next enter a *Repeat* loop which continues until the *Esc* key is entered on the keyboard. The loop awaits a keystroke. When a keystroke is entered, if it is 00, this indicates a special key that outputs two bytes of data, so the second byte is read. Next a *Case* statement takes action depending upon the keystroke. For a left arrow, the cursor location is changed one pixel to the left; right arrow, one pixel to the right; up arrow, one pixel upward; and for the down arrow, the cursor location is changed one pixel downward. Diagonal cursor position changes are also used. For the *Home* key, the cursor position changes one pixel left and one pixel up; *PgUp* key, one pixel right and one pixel up; *End* key, one pixel left and one pixel down; *PgDn* key, one pixel right and one pixel down. Next, the *PutCursor* procedure places the cursor at the new position. The values for the real

and imaginary parts of the Julia set constant are calculated. Next the procedure calls *BlockClearHalfScreen* to clear the half screen of the old Julia set, and ready to draw a new one. This procedure simply uses a pair of nested *For* loops to fill every memory location in the half-screen display memory with a zero, so half of the screen turns black. Next the *CalcJSet* procedure calculates the Julia set using the inverse technique. This procedure initializes the x and y coordinates and enters a *Repeat* loop which reiterates until a key is pressed. Each iteration computes dx and dy , the differences between the current values of x and y and the real and imaginary values of the Julia constant, respectively. Considering these differences as a two-dimensional vector, the vector angle and length are computed. Next, a random number between 0 and 1 is obtained. If it is less than 0.5, the direction of the vector is reversed. The procedure computes new values of x and y and uses them to generate a new position on the screen. The pixel at this new position is plotted in light red. This process continues until a key is pressed. The procedure returns to *WalkAbout* which continues its loop, and permits you to move the cursor and generate a new Julia set. At any time, you can view the values of cx and cy for the Julia set being generated, by hitting the spacebar. The *WalkAbout* procedure terminates when the *Esc* key is pressed, whereupon the procedure returns to the main program. The procedure *ExitGraphics* is called, returning you to the text mode. This completes the program. A typical Mandelbrot/Julia set display is shown in Plate 1.

Simulating Two Orbiting Particles

In this section, look at a program that displays two orbiting particles. Normally, a particle in motion continues to move in a straight line. If, instead, the particle moves in a circle at a constant velocity, there is a changing velocity toward the center of the circle. Changing velocity is acceleration, and this acceleration can be shown to be:

$$a = \frac{v^2}{r}$$

(Equation 3-2)

where a is acceleration, v is the velocity of the particle, and r is the radius of the circular orbit. By Newton's Second Law of Motion, force is equal to mass time

acceleration, so that:

$$F=ma= \frac{mv^2}{r} \quad (\text{Equation 3-3})$$

This is centripetal force. It is a force acting toward the center of the circle. When you see a particle moving in an orbit rather than flying off in a straight line, you'll know that there must be a force acting toward the orbit's center. This is gravitational force. Suppose each object in the system has a position, represented as (x_n, y_n) , and a mass, represented as m_n , where n indicates the number of the object (i.e., $n=1$ for the first object and $n=2$ for the second object). Object positions and the distances between the objects are measured in meters and the mass of each object is measured in kilograms. The distance between the two objects is simply the length of a straight line connecting them. Broken down into Cartesian coordinates, it is:

$$d_x = |x_2 - x_1| \quad (\text{Equation 3-4})$$

$$d_y = |y_2 - y_1| \quad (\text{Equation 3-5})$$

The length of the distance vector is:

$$D = \sqrt{D_x^2 + D_y^2} \quad (\text{Equation 3-6})$$

Now that you know the distance between the objects, you can calculate the gravitational force between them using the Law of Universal Gravitation. This law states that the force due to gravity, F_g , is directly proportional to the product of the masses (m_1 and m_2) and inversely proportional to the distance (d), squared. The proportionality term is the Gravitational Constant of Proportionality (g), and has the value of $6.67E-11$ ($N.m^2/kg^2$). The resulting equation is thus:

$$F_g = g \frac{m_1 m_2}{D^2} \quad (\text{Equation 3-7})$$

This is an attractive force along the direction of the line connecting the two

HOW TO USE THE MODULES

objects. The force is a vector quantity, in that it has a direction and a magnitude. It can also be expressed as a horizontal magnitude and a vertical magnitude (F_x and F_y).

For a system of particles to be in equilibrium, the velocity and distance of each particle from the center of mass must be such that the centripetal force is equal to the gravitational force between the particle and the center of mass. If the gravitational force is too small, the particle will escape orbit; if too large, the particle will be pulled in to the center of mass.

Combining our equations for gravitational force and acceleration, you obtain:

$$a_{x1} = \frac{gm_2}{D^2} \times \frac{D_x}{D} \quad (\text{Equation 3-8})$$

and

$$a_{y1} = \frac{gm_2}{D^2} \times \frac{D_y}{D} \quad (\text{Equation 3-9})$$

Because acceleration is change in velocity of an object with respect to change in time, you obtain the following equations:

$$dv_x = a_x t \quad (\text{Equation 3-10})$$

and

$$dv_y = a_y t \quad (\text{Equation 3-11})$$

If you start with an initial velocity of v_0 , at the end of time t you'll have the new velocity components:

$$v_x = v_{x0} + a_x t \quad (\text{Equation 3-12})$$

and

$$v_y = v_{y0} + a_y t \quad (\text{Equation 3-13})$$

Thus the velocity at the end of the interval is $v_0 + at$. Therefore the average velocity throughout the interval is approximately:

$$v_{avg} = \frac{v_0 + v_0 + at}{2} = v_0 + \frac{at}{2} \quad (\text{Equation 3-14})$$

You'll want your program to show the position of each particle, with continual updating as time passes. Thus you'll want to look for a formula to express the particle's position. Since velocity is the rate of change of position with respect to time, over any interval of time the change in position is the average velocity multiplied by the time. You now have equations that tell will tell you the change in positions of the objects:

$$dx = v_{x0} + \frac{at^2}{2} \quad (\text{Equation 3-15})$$

and

$$dy = v_{y0} + \frac{at^2}{2} \quad (\text{Equation 3-16})$$

Assuming that you know the initial position (x_0, y_0) , then at any time the current position is:

$$x = x_0 + v_{x0}t + \frac{a_x t^2}{2} \quad (\text{Equation 3-17})$$

and

$$y = y_0 + v_{y0}t + \frac{a_y t^2}{2} \quad (\text{Equation 3-18})$$

The *Orbit-2P.Pas* Program

The *Orbit-2P.Pas* program creates a two-part display. Near the center of the screen are two particles, one a red circle and the other a green circle, continually rotating in their orbits with a blue line connecting them. At the top right-hand corner of the screen is a meter comprised of a blue circle showing the velocity vector of the green particle in dark green with its acceleration vector in light green. The velocity

HOW TO USE THE MODULES

of the red particle is in dark red with its acceleration vector in light red. The program is listed in Figure 3-2. When you run the program, the orbit will be continually changing with time, presenting an interesting display. A static display is not very interesting, however, which is why there is no color plate of it.

```
{  
    Orbit-2P.Pas = Two Particle Orbit Simulator  
increase time step size 'dt' to greater than zero to increase  
speed  
}  
  
Uses  
    Dos, Crt;  
  
{$I Math.Inc}  
{$I Graph.Inc}  
  
Var  
    MSize      : Integer;  
    x, y        : Integer;  
    oVx1, oVy1  : Integer;  
    oAx1, oAy1  : Integer;  
    oVx2, oVy2  : Integer;  
    oAx2, oAy2  : Integer;  
  
Procedure InitMeters; { meters for display of velocities and  
                        accelerations }  
  
Begin  
    MSize := 30;  
    Circle(MSize, MSize, MSize - 4, 35);
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
oVx1 := MSize;  
oVy1 := MSize;  
oAx1 := MSize;  
oAy1 := MSize;  
oVx2 := MSize;  
oVy2 := MSize;  
oAx2 := MSize;  
oAy2 := MSize
```

```
End;
```

```
Procedure Meter(dx, dy : Real; Var xt, yt : Integer; Radius,  
Color : Integer);
```

```
Var
```

```
  r : Real;
```

```
Begin
```

```
  Draw(MSize, MSize, xt, yt, 0);  
  r := Radius / Sqrt(Sqr(dx) + Sqr(dy)) - 5;  
  x := Round(dx * r) + MSize;  
  y := -Round(dy * r) + MSize;  
  xt := x;  
  yt := y;  
  Draw(MSize, MSize, x, y, Color)
```

```
End;
```

```
Var
```

```
  X1, Y1, Vx1, Vy1, Ax1, Ay1 : Real;  
  X2, Y2, Vx2, Vy2, Ax2, Ay2 : Real;  
  OneOverDistSqr, dt          : Real;  
  Dx, Dy, Tx, Ty              : Real;  
  M1, M2, Xp1, Yp1, Xp2, Yp2 : Integer;
```

```
Begin
```

```
  M1 := 6; { masses of particles }  
  M2 := 6;
```


HOW TO USE THE MODULES

```
X1 := -40.0; Y1 := 00.0;           { initial positions }
X2 := 40.0; Y2 := 00.0;

Vx1 := 0.000; Vy1 := 1.000; { initial velocities }
Vx2 := 0.000; Vy2 := -1.000;

dt := 0.5;                         { time step size }

InitGraphics;

InitMeters;

Repeat

  Xp1 := 160 + Trunc(X1);
  Yp1 := 100 - Trunc(Y1);
  Xp2 := 160 + Trunc(X2);
  Yp2 := 100 - Trunc(Y2);
                                     { draw particles }
  Draw(Xp1, Yp1, Xp2, Yp2, 35); { blue }

  Circle(Xp1, Yp1, M1, 71); { green }

  Circle(Xp2, Yp2, M2, 143); { red }

  X1 := X1 + Vx1 * dt; { r = r + vt }
  Y1 := Y1 + Vy1 * dt;
  X2 := X2 + Vx2 * dt;
  Y2 := Y2 + Vy2 * dt;

  Dx := X2 - X1; { particle vector distances }
  Dy := Y2 - Y1;

  OneOverDistSqr := 1 / (Sqr(Dx) + Sqr(Dy));
  { one over distance squared since no need for  $(\sqrt{\phantom{x}})^2$  }

  Tx := Dx * OneOverDistSqr; { temporary calculations }
  Ty := Dy * OneOverDistSqr;

  Ax1 := M2 * Tx; { F = ma }
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
Ay1 := M2 * Ty;  
Ax2 := -M1 * Tx;  
Ay2 := -M1 * Ty;  
  
Vx1 := Vx1 + Ax1 * dt;           { v = v + at }  
Vy1 := Vy1 + Ay1 * dt;  
Vx2 := Vx2 + Ax2 * dt;  
Vy2 := Vy2 + Ay2 * dt;  
  
Meter(Vx1, Vy1, oVx1, oVy1, 25, 51);  
                                     { Display V1 dark green }  
Meter(Ax1, Ay1, oAx1, oAy1, 12, 71);  
                                     { Display A1 light green }  
  
Meter(Vx2, Vy2, oVx2, oVy2, 25, 123);  
                                     { Display V2 dark red }  
Meter(Ax2, Ay2, oAx2, oAy2, 12, 143);  
                                     { Display A2 light red }  
  
Draw(Xp1, Yp1, Xp2, Yp2, 0);           { clear particles }  
Circle(Xp1, Yp1, M1, 0);  
Circle(Xp2, Yp2, M2, 0)  
  
Until KeyPressed;  
  
ExitGraphics  
  
End.
```

Figure 3-2. Listing of the *Orbit-2P.Pas* Program

The program initializes the masses of the particles, their initial positions, and their initial velocities. It sets the time step to 0.5. This step must always be positive, and can be increased to cause the particles to move faster or decreased to cause them to move slower. Next, *InitGraphics* sets up the graphics mode. The procedure *InitMeters* initializes the meter part of the display. It draws the initial meter circle and sets up the initial velocity and acceleration coordinates. A *Repeat* loop begins and reiterates until a key is pressed. At this point, *ExitGraphics* returns to the text mode, and the program terminates. Within the loop, the particle positions are converted to

positions on the display screen. A blue line is drawn between them and a circle the size of the mass of each is drawn about the coordinate set in the proper color. Next the new position coordinates of each particle after the time step and the new distances between the particles in the x and y directions are computed. The reciprocal of the distance squared is computed and used to compute the new accelerations and velocities for each particle. Then *Meter* is called four times to display the two velocities and two accelerations in the meter display. This procedure has passed to it the beginning and ending x and y coordinates for the vector, the radius of the circle, and the desired color. The procedure blanks the previous value of the vector by redrawing it in the background color. It uses the input coordinates and the radius to compute the beginning and end points of the vector in display-screen coordinates, draws the new vector in the designated color, and returns to the calling program. Next, the program redraws the two particle circles and their connecting line in the background color to clear them from the screen, and repeats the loop to draw them in their new positions.

The *Orbit-3P.Pas* Program

When three particles are involved, the math is similar to the above, for three particles, but a bit more complex because the interactions between pairs of particles must be taken into account. The equations are all similar to those already discussed for the two particle case. The *Orbit-3P.Pas* program creates a display that has two parts. Near the center of the screen are three particles, a red, a white, and a green circle, continually rotating in their orbits, with blue lines connecting each pair of particles. At the top right-hand corner of the screen is a meter comprised of a blue circle showing the velocity vector of the green particle in dark green with its acceleration vector in light green, the red particle in dark red with its acceleration vector in light red, and the white particle in gray with its acceleration in white. The program is listed in Figure 3-3.

The program initializes the particle masses, their initial positions, and their initial velocities. It sets the time step to 0.4: This step must always be positive, but can be increased to move the particles faster or decreased to move them slower. Next, *InitGraphics* sets up the graphics mode. The procedure *InitMeters* initializes the meter

part of the display. It draws the initial meter circle and sets up the initial velocity and acceleration coordinates. Next, a *Repeat* loop begins and reiterates until a key is pressed. At this point, *ExitGraphics* returns to the text mode, and the program terminates. Within the loop, the particle positions convert to positions on the display screen. A blue line is drawn between each pair of particles and a circle the size of the mass of each is drawn about the coordinate set in the proper color. Next, the new position coordinates of each particle after the time step and the new distances between the particles in the x and y directions are computed. The reciprocal of the distance squared is computed and computes the new accelerations and velocities for each particle. Then *Meter* is called six times to display the three velocities and three accelerations in the meter display. This procedure has passed to it the beginning and ending x and y coordinates for the vector, the radius of the circle, and the desired color. The procedure blanks the previous value of the vector by redrawing it in the background color. It uses the input coordinates and the radius to compute the beginning and end points of the vector in display-screen coordinates, draws the new vector in the designated color, and returns to the calling program. Next, the program redraws the three particle circles and their connecting lines in the background color to clear them from the screen, and repeats the loop to draw them in their new positions.

```
{
```

```
Orbit-3P.Pas = Three Particle Orbit Simulator
```

```
increase time step size 'dt' to greater than zero to increase speed
```

```
}
```

```
Uses
```

```
Dos, Crt;
```

```
{ $I Math.Inc }
```


HOW TO USE THE MODULES

```
{ $I Graph.Inc }
```

```
Var
```

```
    MSize      : Integer;  
    x, y       : Integer;  
    oVx1, oVy1 : Integer;  
    oAx1, oAy1 : Integer;  
    oVx2, oVy2 : Integer;  
    oAx2, oAy2 : Integer;  
    oVx3, oVy3 : Integer;  
    oAx3, oAy3 : Integer;
```

```
Procedure InitMeters; { meters for display of velocities and  
accelerations }
```

```
Begin
```

```
    MSize := 30;  
    Circle(MSize, MSize, MSize - 4, 35);  
    oVx1 := MSize;  
    oVy1 := MSize;  
    oAx1 := MSize;  
    oAy1 := MSize;  
    oVx2 := MSize;  
    oVy2 := MSize;  
    oAx2 := MSize;  
    oAy2 := MSize;  
    oVx3 := MSize;  
    oVy3 := MSize;  
    oAx3 := MSize;  
    oAy3 := MSize
```

```
End;
```

```
Procedure Meter(dx, dy : Real; Var xt, yt : Integer; Radius,  
Color : Integer);
```

```
Var
```

```
    r : Real;
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

Begin

```
Draw(MSize, MSize, xt, yt, 0);  
r := Radius / Sqrt(Sqr(dx) + Sqr(dy)) - 10;  
x := Round(dx * r) + MSize;  
y := -Round(dy * r) + MSize;  
xt := x;  
yt := y;  
Draw(MSize, MSize, x, y, Color)
```

End;

Var

```
X1, Y1, Vx1, Vy1, Ax1, Ay1      : Real;  
X2, Y2, Vx2, Vy2, Ax2, Ay2      : Real;  
X3, Y3, Vx3, Vy3, Ax3, Ay3      : Real;  
D12, D23, D31, dt               : Real;  
Dx12, Dx23, Dx31                : Real;  
Dy12, Dy23, Dy31                : Real;  
Tx12, Tx23, Tx31                : Real;  
Ty12, Ty23, Ty31                : Real;  
Xp1, Yp1, Xp2, Yp2, Xp3, Yp3    : Integer;  
M1, M2, M3                       : Integer;
```

Begin

```
M1 := 3;                               { masses of particles }  
M2 := 8;  
M3 := 6;
```

```
X1 := -50;    Y1 := 0;                 { initial positions }  
X2 := 0;      Y2 := 0;  
X3 := 50;     Y3 := 0;
```

```
Vx1 := 0.1000; Vy1 := 0.1000;         { initial velocities }  
Vx2 := 0.0010; Vy2 := 0.0028;  
Vx3 := -0.1010; Vy3 := -0.3000;
```

```
dt := 0.4;                               { time step size }
```


HOW TO USE THE MODULES

InitGraphics;

InitMeters;

Repeat

Xp1 := 160 + Trunc(X1);

Yp1 := 100 - Trunc(Y1);

Xp2 := 160 + Trunc(X2);

Yp2 := 100 - Trunc(Y2);

Xp3 := 160 + Trunc(X3);

Yp3 := 100 - Trunc(Y3);

{ draw particles }

Draw(Xp1, Yp1, Xp2, Yp2, 35);

Draw(Xp2, Yp2, Xp3, Yp3, 35);

Draw(Xp3, Yp3, Xp1, Yp1, 35);

Circle(Xp1, Yp1, M1, 143);

Circle(Xp2, Yp2, M2, 71);

Circle(Xp3, Yp3, M3, 251);

X1 := X1 + Vx1 * dt; { r = r + vt }

Y1 := Y1 + Vy1 * dt;

X2 := X2 + Vx2 * dt;

Y2 := Y2 + Vy2 * dt;

X3 := X3 + Vx3 * dt;

Y3 := Y3 + Vy3 * dt;

Dx12 := X1 - X2; { particle vector distances }

Dy12 := Y1 - Y2;

Dx23 := X2 - X3;

Dy23 := Y2 - Y3;

Dx31 := X3 - X1;

Dy31 := Y3 - Y1;

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
{one over distance^1.5}
D12 := Sqrt( Sqr(Dx12) + Sqr(Dy12) );
D12 := 1 / (D12 * D12 * D12);

D23 := Sqrt( Sqr(Dx23) + Sqr(Dy23) );
D23 := 1 / (D23 * D23 * D23);

D31 := Sqrt( Sqr(Dx31) + Sqr(Dy31) );
D31 := 1 / (D31 * D31 * D31);

Tx31 := Dx31 * D31;      { temporary calculations }
Ty31 := Dy31 * D31;

Tx12 := Dx12 * D12;
Ty12 := Dy12 * D12;

Tx23 := Dx23 * D23;
Ty23 := Dy23 * D23;

Ax1 := ( M3 * Tx31 - M2 * Tx12 );   { F = ma }
Ay1 := ( M3 * Ty31 - M2 * Ty12 );

Ax2 := ( M1 * Tx12 - M3 * Tx23 );
Ay2 := ( M1 * Ty12 - M3 * Ty23 );

Ax3 := ( M2 * Tx23 - M1 * Tx31 );
Ay3 := ( M2 * Ty23 - M1 * Ty31 );

Vx1 := Vx1 + Ax1 * dt;               { v = v + at }
Vy1 := Vy1 + Ay1 * dt;

Vx2 := Vx2 + Ax2 * dt;
Vy2 := Vy2 + Ay2 * dt;

Vx3 := Vx3 + Ax3 * dt;
Vy3 := Vy3 + Ay3 * dt;

Meter(Vx1, Vy1, oVx1, oVy1, 25, 123); {Display V1 dark red}
Meter(Ax1, Ay1, oAx1, oAy1, 12, 143);
                                     {Display A1 light red}
```


HOW TO USE THE MODULES

```
Meter(Vx2, Vy2, oVx2, oVy2, 25, 51);  
                                     {Display V2 dark green}  
Meter(Ax2, Ay2, oAx2, oAy2, 12, 71);  
                                     {Display A2 light green}  
  
Meter(Vx3, Vy3, oVx3, oVy3, 25, 231);  
                                     {Display V3 dark gray}  
Meter(Ax3, Ay3, oAx3, oAy3, 12, 251);  
                                     {Display A3 light gray}  
  
Draw(Xp1, Yp1, Xp2, Yp2, 0);      { clear particles }  
Draw(Xp2, Yp2, Xp3, Yp3, 0);  
Draw(Xp3, Yp3, Xp1, Yp1, 0);  
Circle(Xp1, Yp1, M1, 0);  
Circle(Xp2, Yp2, M2, 0);  
Circle(Xp3, Yp3, M3, 0)  
  
Until KeyPressed;  
  
ExitGraphics
```

End.

Figure 3-3. The *Orbit-3P.PAS* Program

Generating a Sierpinski Triangle

The Sierpinski triangle is an interesting fractal figure. Aside from the fact that it creates a beautiful and captivating display, there seem to be more ways of generating it than for almost any other fractal. Many of these techniques seem to have very different starting points, yet, when plotted with enough levels of detail the resulting figure seems to be the same in every case. In *Fractal Programming in Turbo Pascal* (M & T Books, 1990) several ways to generate this triangle are shown. Here is another method. This method makes use of a recursive technique.

```
{  
  Triangle.Pas = Recursive Triangle Generator  
}
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

Uses

Dos, Crt;

{ \$I Math.Inc }

{ \$I Graph.Inc }

Const

Ang1 = 0; { 90 }

Ang2 = 120; { 210 }

Ang3 = 240; { 330 }

Var

X1, Y1, X2, Y2, X3, Y3 : Integer;

Level, Xc, Yc : Integer;

Col : Integer;

Procedure Triangle(X1, Y1, X2, Y2, X3, Y3, Level : Integer);

Var

Xp1, Xp2, Xp3 : Integer;

Yp1, Yp2, Yp3 : Integer;

Begin

If (Level = 0) Then

Begin

Xp1 := Xc + X1;

Yp1 := Yc - Y1;

Xp2 := Xc + X2;

Yp2 := Yc - Y2;

Xp3 := Xc + X3;

Yp3 := Yc - Y3;

Draw(Xp1, Yp1, Xp2, Yp2, Col);

Draw(Xp2, Yp2, Xp3, Yp3, Col);

Draw(Xp3, Yp3, Xp1, Yp1, Col)

End

Else

HOW TO USE THE MODULES

```
Begin
  Triangle(X1, Y1, (X1 + X2) Div 2, (Y1 + Y2) Div 2,
    (X1 + X3) Div 2, (Y1 + Y3) Div 2, Level - 1);
  Triangle(X2, Y2, (X2 + X3) Div 2, (Y2 + Y3) Div 2,
    (X2 + X1) Div 2, (Y2 + Y1) Div 2, Level - 1);
  Triangle(X3, Y3, (X3 + X1) Div 2, (Y3 + Y1) Div 2,
    (X3 + X2) Div 2, (Y3 + Y2) Div 2, Level - 1);
  Col := Level * 36 + 35
End
End;
```



```
Begin
  Title;
  Level := 0;
  WriteLn('Recursive Triangle Generator');
  WriteLn;
  Write('Level => ');
  ReadLn(Level);
  InitGraphics;
  Col := 35;

  Xc := 160;
  Yc := 100;

  X1 := Round(Xc * CosD(Ang1));
  Y1 := Round(Yc * SinD(Ang1));
  X2 := Round(Xc * CosD(Ang2));
  Y2 := Round(Yc * SinD(Ang2));
  X3 := Round(Xc * CosD(Ang3));
  Y3 := Round(Yc * SinD(Ang3));

  Triangle(X1, Y1, X2, Y2, X3, Y3, Level);

  ExitGraphics
End.
```

Figure 3-4. Listing of Program to Draw a Sierpinski Triangle

At the lowest level, the program calls a procedure *Triangle*, which draws a triangle. At higher levels, *Triangle* draws nothing. Instead, it calls itself three times at the next lower level for three half-sized triangles along the same sides and having the same apexes as the original triangle. This process repeats until triangles of the lowest level are reached. These are actually drawn. The program is listed in Figure 3-4, and the resulting Sierpinski triangle is shown in Plate 2.

The program displays the legend "Recursive Triangle Generator," followed by "Level=>". It reads in the level selected by the user. Next the procedure *InitGraphics* sets up the graphics display mode. The initial color is set and two coordinate parameters, *Xo* and *Yo*, are initialized. These parameters, with a set of trigonometric operations, create the three sets of coordinates for a large equilateral triangle. The program calls *Triangle* for these coordinates and the specified level. If the level is zero, this procedure draws a line in the specified color between each pair of coordinates to create a triangle on the screen. If the level is greater than zero, instead of drawing a triangle, the procedure calls itself three times. Each call has as its coordinate parameters one of the three vertices of the current triangle plus the points halfway along the sides adjoining that vertex. Also, the level parameter is reduced by one. The color is changed and the procedure terminates. Thus a recursive process occurs, with more and more calls to *Triangle* until the zeroeth level is reached, with each call drawing a triangle on the screen.

The Bifurcation Diagram

One of the first examples of chaotic behavior discovered in the sciences was when scientists looked closely at the population equation, which supposedly models the behavior of population growth. This is an iterated equation of the form:

$$x_{n+1} = rx_n(1 - x_n) \quad (\text{Equation 3-19})$$

The bifurcation diagram is a graph created by selecting different values of *r* and performing enough iterations for each value so that the equation becomes a steady set of values (if it is ever going to). Then the results of a number of iterations are plotted to show whether the equation settles to one value, is cycling between two or

more values, or has a different value for each iteration. A somewhat different form of the equation is used to obtain the bifurcation diagram, namely:

$$x_n = r \left(\frac{x_{n-1}}{(1 - x_{n-1})^b} \right) \quad (\text{Equation 3-20})$$

The program to generate a bifurcation diagram from this equation is listed in Figure 3-5. A picture of the final plot is shown in Plate 3. The program begins by listing a number of constants and their variables. It calls *InitGraphics* to set up the graphics display mode and initializes several variables. Next, a *Repeat* loop begins, with *lambda* at the minimum value (0). This loop iterates, with *lambda* increasing by *dlambda* at each iteration, until the maximum value (125) is reached, whereupon the loop terminates. The loop sets *x* to the initial value (0.9). It begins a *For* loop which iterates 300 times. On each pass through the loop the iterated equation is solved. The proper pixel values corresponding to the two coordinates are computed. To give the equation a chance to settle down to its final values, the first 64 iterations are not plotted. After this, the program plots a red pixel at the proper coordinates at each iteration of the equation. When the *For* loop is complete, *lambda* is increased to the next value and the process repeated. When the final value of *lambda* terminates the *Repeat* loop, the program calls *ExitGraphics* to return to the text display mode and terminates.

```
{
```

```
Bifurcat.Pas = Bifurcation Diagram Generator
```

```
}
```

```
Uses
```

```
Dos, Crt;
```

```
{$I Math.Inc}
```

```
{$I Graph.Inc}
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
Const
    min    = 0;
    max    = 125;
    N      = 300;
    Res    = 200;
    beta   = 5;
    x0     = 0.9;
    T      = 25;

Var
    dlambda : Real;
    lambda   : Real;
    scale, x : Real;
    i, Temp  : Integer;
    xxx, yyy : Integer;

Begin
    InitGraphics;

    dlambda := (max - min) / Res;
    scale := Res / max;
    lambda := min;

    Repeat
        x := x0;
        yyy := Round(lambda * scale);
        For i := 1 To N Do
            Begin
                x := lambda * x / Power( (1+x), (beta) );
                xxx := Round(30 * x);
                If i > 64 Then
                    Plot(xxx, yyy, Temp+128)
            End;
            lambda := lambda + dlambda
        Until (lambda > max);

    ExitGraphics
End.
```

Figure 3-5. Listing of Program to Generate Bifurcation Diagram

Creating a Strange Attractor

This program generates a two-dimensional projection of the three-dimensional strange attractor that illustrates the system represented by the following set of iterated equations:

$$x_{n+1} = \sin(ay_n) - z_n \cos(bx_n) \quad (\text{Equation 3-21})$$

$$y_{n+1} = z_n \sin(cx_n) - \cos(dy_n) \quad (\text{Equation 3-22})$$

$$z_{n+1} = \sin(x_n) \quad (\text{Equation 3-23})$$

The program listed in Figure 3-6 creates this strange attractor and uses the procedures that have already been developed to project it onto the two-dimensional screen. The resulting display is shown in Plate 4. There are a number of constants at the beginning of this program, including the coefficients of the equation set and the minimum and maximum values for each of the three coordinates.

The program begins with an initialization process that includes calling *InitPerspective* and *InitPlotting* (which were described in Chapter 2) and *InitGraphics* to go into the graphics display mode. Next, *PutAxisPalette* sets up the mode where the axes and palette are displayed on the screen. Next, *AxisAndPalette* is called to display the palette and the axes of the coordinate system. The factors to convert x , y , and z into screen pixel coordinates are set up. The three coordinates are initialized to zero. The program begins a *Repeat* loop which continues until a key is pressed. Thus the display continues to add points and become more dense until the user terminates it. At each iteration of the loop, the program generates a new set of x , y , and z values from the previous set. It converts these to three-dimensional screen pixel coordinates. It generates a color value, using the value of the pixel coordinate in the x dimension. The program goes to the coordinate location on the screen and retrieves the color of the pixel there, using the procedure *GetPixel3D*. This procedure first calls *MapCoordinates* to create a pair of two-dimensional screen coordinates from the three-dimensional ones. It calls *GetPixel* to read the color of the pixel at that screen location. Then, if the color that was generated is a higher number than the

color already at the pixel, this color is plotted to the screen. At first, it may not be obvious what is happening. The color is selected so that its number increases with the increasing of the x coordinate in the three-dimensional system. When three-dimensional coordinates are projected onto the two-dimensional screen, it may turn out that several three-dimensional locations end up at the same two-dimensional position. If this occurs, you'll want the color of that two-dimensional position to be that of the nearest three-dimensional position, since that is the color you would see. That nearest position is the one with the highest value of the three-dimensional x coordinate, so the program lines just described assure that the screen is painted with the right color. This completes the loop; it iterates until a key is pressed, continuously adding complexity to the display. When the key is pressed, the procedure *ExitGraphics* returns to the text display mode. The program terminates.

```
{
  3DStrAttr.Pas = Three-Dimensional Strange Attractor
                    Generator
}
```

```
Uses
  Dos, Crt;
```

```
{$I Math.Inc}
```

```
{$I Graph.Inc}
```

```
Function GetPixel3D(x, y, z : Integer) : Integer;
```

```
  Var
    xp, yp : Integer;
```

```
  Begin
    MapCoordinates(x, y, z, xp, yp);
```


HOW TO USE THE MODULES

```
GetPixel3D := GetPixel(xp, yp)
End;
```

Const

```
xxmin = -2;
xxmax = 2;
yymin = -2;
ymax = 2;
zzmin = -2;
zzmax = 2;
res = 200;
a = 2.24;
b = 0.43;
c = -0.65;
d = -2.43;
e = 1.00;
```

Var

```
xinc, yinc, zinc      : Real;
x, y, z, xx, yy, zz   : Real;
xxx, yyy, zzz, col, pix : Integer;
```

Begin

```
InitPerspective(False,0,0,500,500);
InitPlotting(240,18);
InitGraphics;
PutAxisAndPalette(True);
AxisAndPalette;
```

```
xinc := res / (xxmax - xxmin);
yinc := res / (ymax - ymin);
zinc := res / (zzmax - zzmin);
x := 0;
y := 0;
z := 0;
```

Repeat

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
xx := sin( a * y ) - z * cos( b * x );
yy := z * sin( c * x ) - cos( d * y );
zz := e * sin( x );
x := xx;
y := yy;
z := zz;
xxx := Round( xx * xinc );
yyy := Round( yy * yinc );
zzz := Round( zz * zinc );
col := (xxx + xres div 2) mod 251;
pix := GetPixel3D(xxx, yyy, zzz);
If (col > pix) Then
    CartesianPlot3D(xxx, yyy, zzz, col)
Until KeyPressed;
```

```
ExitGraphics
End.
```

Figure 3-6. Listing of Program to Generate a Strange Attractor

Generating the Lorenz Attractor

The Lorenz attractor began as a simplified model for long range weather predictions. It consists of the following iterated equations:

$$\frac{dx}{dt} = 10(y - x) \quad (\text{Equation 3-24})$$

$$\frac{dy}{dt} = xz + 28x - y \quad (\text{Equation 3-25})$$

$$\frac{dz}{dt} = xy - \frac{8}{3}z \quad (\text{Equation 3-26})$$

The program to generate the Lorenz attractor is listed in Figure 3-7 and the resulting figure shown in Plate 5. The program begins with the familiar initialization process in which the procedures *InitPerspective*, *InitPlotting*, and *InitGraphics* are called.

HOW TO USE THE MODULES

```
{  
  3DLorenz.Pas = Three-Dimensional Lorenz Attractor Generator  
}
```

```
Uses  
  Dos, Crt;
```

```
{$I Math.Inc}
```

```
{$I Graph.Inc}
```

```
Const  
  Sc = 2.2;  
  EightThirds = 2.6666667;
```

```
Var  
  x, y, z      : Real;  
  T, T9        : Real;  
  d1, d2, d3   : Real;  
  Step, Stp    : Real;  
  Xo, Yo, Zo   : Integer;  
  Xn, Yn, Zn   : Integer;  
  Col          : Integer;  
  Pt, Tmp      : TDA;
```

```
Procedure DrawLine;  
Begin  
  Xn := Round(Sc * X);  
  Yn := Round(Sc * Y);  
  Zn := Round(Sc * Z);
```

```
  If (T9 = 0) Then  
    Begin
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
        Xo := Xn;  
        Yo := Yn;  
        Zo := Zn;  
        Vec(Xo, Yo, Zo, Tmp)  
    End;
```

```
    Col := Round(T9) Mod 25 + 128;
```

```
    Vec(Xn, Yn, Zn, Pt);
```

```
    DrawLine3D(Tmp, Pt, Col);
```

```
        Xo := Xn;  
        Yo := Yn;  
        Zo := Zn;  
        Tmp := Pt  
    End;
```

```
Begin
```

```
    InitPerspective(False, 0, 0, 500, 500);  
    InitPlotting(-45,35);  
    InitGraphics;  
    PutAxisAndPalette(True);  
    AxisAndPalette;
```

```
    X := 5.0;  
    Y := 5.0;  
    Z := 5.0;
```

```
    Col := 0;  
    Stp := 0.01;  
    Step := Stp / 25;  
    T9 := 0;
```

```
    Repeat  
        T := T9;  
        Repeat  
            D1 := 10 * (Y - X);
```



```

D2 := 28 * X - Y - X * Z;
D3 := X * Y - EightThirds * Z;
X := X + D1 * Step;
Y := Y + D2 * Step;
Z := Z + D3 * Step;
T := T + Step
Until (T >= T9 + Stp);
DrawLine;
T9 := T9 + Stp
Until (T9 >= 500) Or KeyPressed;

ExitGraphics

```

End.

Figure 3-7. Listing of Program to Generate the Lorenz Attractor

The two procedures are called which result in a palette and a set of cartesian coordinate axes being displayed, as described in the previous section. The initial coordinate value is set to (0.5,0.5,0.5) and the step, time step, and color are set to their initial values. The differential equations are solved for a time increment, which is one-twenty-fifth of a step, and the time and coordinate values have the appropriate increments added to them. This is repeated 25 times to complete one step. The program calls *DrawLine* to draw a line between the beginning and ending points. This entire process is repeated until 500 complete time steps have occurred or until a key is pressed, whereupon *ExitGraphics* returns us to the text display mode and terminates. The procedure *DrawLine* scales the current coordinate values and converts them to integers. If the time indicates that this is the first pass, the coordinates are converted to a vector, *Tmp*, which is the line beginning, and the procedure returns to the calling program. On all additional passes, the color is computed, based on the elapsed time, and the coordinates are converted to a vector, *Pt* that represents the end of the line. *DrawLine3D*, described in the previous chapter, converts to two-dimensional screen coordinates and draws the line on the screen. The line-ending point becomes the line-beginning point for the next pass through this procedure.

Part II

CHAPTER 4

Solid Modeling Theory and Database Structure

WIRE FRAME AND SOLID MODELING

Before you jump right in to learning how to build your first model, you need to understand the difference between wire frame and solid modeling. Wire frame modeling is a technique where you define the edges of a 3D object, but you don't define the surfaces. Solid modeling, on the other hand, allows you to define the surfaces of a 3D object, as well as the edges. This means that you can create a solid model of a 3D object, and you can calculate its volume, mass, and other properties. Wire frame modeling is typically used for creating 3D models of objects that are transparent or where the internal structure is important. Solid modeling is typically used for creating 3D models of objects that are opaque and where the external shape is important.

The first step in creating a solid model is to define the edges of the object. This is done by creating a wire frame model. Once the wire frame is created, you can then define the surfaces of the object. This is done by creating a solid model. The solid model is then used to calculate the volume, mass, and other properties of the object. The first step in creating a solid model is to define the edges of the object. This is done by creating a wire frame model. Once the wire frame is created, you can then define the surfaces of the object. This is done by creating a solid model. The solid model is then used to calculate the volume, mass, and other properties of the object.

CHAPTER 4

Solid Modeling Theory and Database Structure

There are many different ways in which a computer generates a scene having realistic object shading and reflections. The method used in this section subdivides each object into a number of small facets. A facet is defined by a number of vertices. You usually want to use four vertices to define a facet. The surface created by connecting pairs of facets together is assigned a shade of color that depends upon the light impinging upon it and its inherent color. This surface is translated to two-dimensional screen coordinates and painted in the proper shade on the screen. Additionally, it is possible to define the reflections of objects in the same way. By properly selecting which reflections to draw first and by drawing the real objects last, you can produce a scene in which reflections are properly overlaid by the real objects.

Before you jump right in to learning how to break objects into facets and how to handle the resultant data, you need to make some decisions about the kind and amount of data permitted. Mostly, this is a function of the available storage space in Turbo Pascal. As you delve into this, you'll see that you can handle quite a few facets, so you can describe objects of considerable complexity if the need arises. The newest versions of Turbo Pascal have techniques for using extended memory if your computer has it, and if you need gigantic data bases, but that subject won't be discussed here. It is something for you to investigate if your interests lie in that direction.

The data structure containing the facets of a scene is part of a file called *Model.Inc* included in all the solid-modeling programs. This file also contains routines to clear, load, and save vertex data and to draw a wire facet or paint a solid facet on the screen. The file, listed in Figure 4-1, is described in the paragraphs that follow.

Scale Factors

The first action in this section of code is that *ScaleData* is initialized to 23000. This is a scale factor that converts real positional data to integers so faster processing occurs. The maximum value of an integer is 32767, and if you have a coordinate having the maximum value of 23000, rotated 45 degrees, it may take on a value of approximately 32767. Thus this scaling value assures that you cannot exceed integer bounds with any rotation. Next, *ScaleImage* is set to 100. This is the right factor for converting the integer system used to pixel coordinates on the screen.

Vertex and Facet Arrays

Each of the objects that are included in a scene has its facet and vertex information stored in a separate disk file. When you paint the object on the screen, this information transfers to arrays in the program. This places a constraint on the number of facets and vertices in an object description, since the information must fit into the available memory.

Begin with the constraint that the available memory for data is 65,536 (64K) bytes. Within this capacity is the reflection buffer, which needs one bit for every pixel on the screen. You will be using the 320-by-200 pixel display mode, so you'll need 64,000 bits, or 8,000 bytes for this storage task. The object buffer is next, but you'll look at this in more detail later. For the moment, say it contains a maximum of 40 objects and that each object description requires 70 bytes, so the total memory requirement is 2,800 bytes. Subtracting these from the total memory available, 54,736 bytes maximum are available for vertex and facet information. Call this maximum M_x .

```
{  


Variable Declarations

  
}
```


SOLID MODELING THEORY AND DATABASE STRUCTURE

Const

NumVerticesInFacet = 4; { 4 vertices for 4 sided facets }

MaxVertexNumInFacet = NumVerticesInFacet;

ScaleData = 23000;

{ approx 32767 / Sqrt(2) - because of 45° rot

ScaleImage = 100; { scale for display }

(*

MaxFacet = 1678; { independant : MaxFacet = n }

MaxVertex = 6712; { vertices MaxVertex = 4 * n }

*)

MaxFacet = 3600; { shared : MaxFacet = (n-1) * (n-1) }

MaxVertex = 3721; { vertices MaxVertex = n * n }

Var

LastVertex, LastFacet : Integer;

LastVertexNumInFacet : Integer;

VertexNum, FacetNum : Integer;

VertexNumInFacet : Integer;

Vertex : Array[1..MaxVertex] Of TDIA;

{ LastVertex vertices }

Facet : Array[1..MaxFacet, 1..MaxVertexNumInFacet] Of Integer;

{ LastFacet facets - each having LastVertexNumInFacet
vertices }

{

Routines to Load and Save Vertex and Facet Data

InitVertexBuffer - clear memory for facet and vertex data

LoadData - load facet and vertex data

SaveData - save facet and vertex data

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
}
```

```
Var  
    InvScaleData, InvScaleImage : Real;
```

```
Procedure InitVertexBuffer;
```

```
Begin
```

```
    InvScaleData := 1 / ScaleData;
```

```
    InvScaleImage := 1 / ScaleImage;
```

```
    For VertexNum := 1 To MaxVertex Do
```

```
        VecNullInt(Vertex[VertexNum]);
```

```
    For FacetNum := 1 To MaxFacet Do
```

```
        For VertexNumInFacet := 1 To MaxVertexNumInFacet Do
```

```
            Facet[FacetNum, VertexNumInFacet] := 0
```

```
        End;
```

```
Type
```

```
    Name = String[12];
```

```
Var
```

```
    DiskFile : File Of Char;
```

```
    TextDiskFile : Text;
```

```
    FileName : Name;
```

```
Procedure LoadData(FileName : Name);
```

```
Var
```

```
    Temp : TDIA;
```

```
    x, y, z : Integer;
```

```
    MSB, LSB : Char;
```

```
Begin
```

```
    Assign(TextDiskFile, FileName);
```

```
    Reset(TextDiskFile);
```

```
    ReadLn(TextDiskFile, LastVertex, LastFacet,
```


SOLID MODELING THEORY AND DATABASE STRUCTURE

```
                                LastVertexNumInFacet);
For VertexNum := 1 To LastVertex Do
  Begin
    Read(TextDiskFile, MSB);
    Read(TextDiskFile, LSB);
    x := Ord(MSB) ShL 8 + Ord(LSB);

    Read(TextDiskFile, MSB);
    Read(TextDiskFile, LSB);
    y := Ord(MSB) ShL 8 + Ord(LSB);
    Read(TextDiskFile, MSB);
    Read(TextDiskFile, LSB);
    z := Ord(MSB) ShL 8 + Ord(LSB);

    If (Abs(x) > ScaleData) Or (Abs(y) > ScaleData) Or
      (Abs(z) > ScaleData) Then
      Begin
        WriteLn('ERROR: data out of range - Vertex #',
              VertexNum);
        Halt
      End
    Else
      VecInt(x, y, z, Vertex[VertexNum]);
  End;

For FacetNum := 1 To LastFacet Do
  For VertexNumInFacet := 1 To LastVertexNumInFacet Do
    Begin
      Read(TextDiskFile, MSB);
      Read(TextDiskFile, LSB);
      Facet[FacetNum, VertexNumInFacet] := Ord(MSB) ShL 8 +
                                             Ord(LSB)
    End;
  Close(TextDiskFile)
End;

Procedure SaveData(FileName : Name);
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
Var
  x, y, z : Integer;

Begin
  Assign(TextDiskFile,FileName);
  Rewrite(TextDiskFile);
  WriteLn(TextDiskFile,LastVertex,' ',LastFacet,' ',
                                                LastVertexNumInFacet);
  For VertexNum := 1 To LastVertex Do
    Begin
      UnVecInt(Vertex[VertexNum], x, y, z);
      Write(TextDiskFile, Chr(x ShR 8));      { Write MSB }
      Write(TextDiskFile, Chr(x And 255));    { Write LSB }
      Write(TextDiskFile, Chr(y ShR 8));      { Write MSB }
      Write(TextDiskFile, Chr(y And 255));    { Write LSB }
      Write(TextDiskFile, Chr(z ShR 8));      { Write MSB }
      Write(TextDiskFile, Chr(z And 255));    { Write LSB }
    End;
  For FacetNum := 1 To LastFacet Do
    For VertexNumInFacet := 1 To LastVertexNumInFacet Do
      Begin
        Write(TextDiskFile, Chr(Facet[FacetNum,
                                                VertexNumInFacet] ShR 8));
        Write(TextDiskFile, Chr(Facet[FacetNum,
                                                VertexNumInFacet] And 255))
      End;
    Close(TextDiskFile)
  End;
```

```
Type
  PixelVector = Record
    x, y : Integer
  End;
  PixelArray  = Array[1..(NumVerticiesInFacet+1)] Of PixelVector;
  VoxelArray  = Array[1..NumVerticiesInFacet] Of TDA;
```

{

SOLID MODELING THEORY AND DATABASE STRUCTURE

Draw a Wire Facet

PutWireFacet - draws a wire outline of a facet
}

Procedure PutWireFacet(Pts : PixelArray; Color : Byte);

Var

t : Integer;

Begin

For t := 1 To 4 Do

Draw(Pts[t].x, Pts[t].y, Pts[t+1].x, Pts[t+1].y, Color)

End;

{

Place a Solid Facet

PutFacet - Place a Facet
}

Procedure PutFacet(Face2d : PixelArray; Color, Intens : Byte);

Var

mnx, mxx, yc : Integer;

Procedure DrawHorizLine;

Var

xc : Integer;

Begin

For xc := mnx To mxx Do

PutPixel(xc, yc, Color, Intens)

End;

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
Var
  i, x      : Integer;
  OldVertNum : Integer;
  VertNum   : Integer;
  mny, mxy  : Integer;
  slope     : Real;

Begin
  mny := Face2d[1].y;
  mxy := Face2d[1].y;
  For i := 2 To LastVertexNumInFacet Do {find min and max y}
                                         {coords}
    Begin
      If (Face2d[i].y > mxy) Then mxy := Face2d[i].y;
      If (Face2d[i].y < mny) Then mny := Face2d[i].y;
    End;
  If (mny < 0) Then mny := 0;
  If (mxy > MaxY) Then mxy := MaxY;
  For yc := mny To mxy Do
    Begin
      mnx := MaxX + 1;
      mxx := -1;
      OldVertNum := LastVertexNumInFacet;
      For VertNum := 1 To LastVertexNumInFacet Do
        Begin
          If (Face2d[OldVertNum].y >= yc) Or
             (Face2d[VertNum].y >= yc) Then
            If (Face2d[OldVertNum].y <= yc) Or
               (Face2d[VertNum].y <= yc) Then
              If Not(Face2d[OldVertNum].y =
                     Face2d[VertNum].y) Then
                Begin
                  slope := (Face2d[VertNum].x -
                           Face2d[OldVertNum].x) /
                           (Face2d[VertNum].y -
                           Face2d[OldVertNum].y);
                  x := Round(slope * (yc -
                                   Face2d[OldVertNum].y)) +
                        Face2d[OldVertNum].x;
                  If (x < mnx) Then mnx := x;
```


SOLID MODELING THEORY AND DATABASE STRUCTURE

```

                                If (x > mxx) Then mxx := x
                                End;
                                OldVertNum := VertNum
                                End;
                                If (mnx < 0) Then mnx := 0;
                                If (mxx > MaxX) Then mxx := MaxX;
                                If (mnx <= mxx) Then DrawHorizLine
                                End
                                End;

```

Figure 4-1. Listing of *Model.inc* File

For a single facet, you have four vertices, each needing description by an integer. The facet also has to be described with four integers, one identifying each of the vertices of which it is comprised. If you have n facets, and each must be described independently, you have a storage requirement of:

$$4nx3 + nx4 = \frac{M_x}{2} \quad (\text{Equation 4-1})$$

The maximum memory available is divided by two because each integer requires two bytes of storage. You have space for approximately 1,678 facets and 6,712 vertices, maximum. Fortunately, this is a worst-case situation. Most of the objects you deal with have facets that share vertices. Look at Figure 4-2, a surface composed of 9 facets and 16 shared vertices. If you generalize this, you have n^2 vertices and $(n-1)^2$ facets. The memory storage requirement therefore is :

$$n^2 \times 3 + (n-1)^2 \times 4 = \frac{M_x}{2} \quad (\text{Equation 4-2})$$

This gives you enough storage for approximately 3721 facets and 3844 vertices. In the listing, the vertex and facet maximums for independent facets are commented out. Reduce n by one in specifying the maximum number of facets, *MaxFacet* (3600), and the maximum number of vertices, *MaxVertex* (3721), to give a little cushion of memory space. The *Vertex* array consists of 3721 sets of four three-dimensional integer vectors, and the *Facet* array consists of 3600 sets of four integers per set.

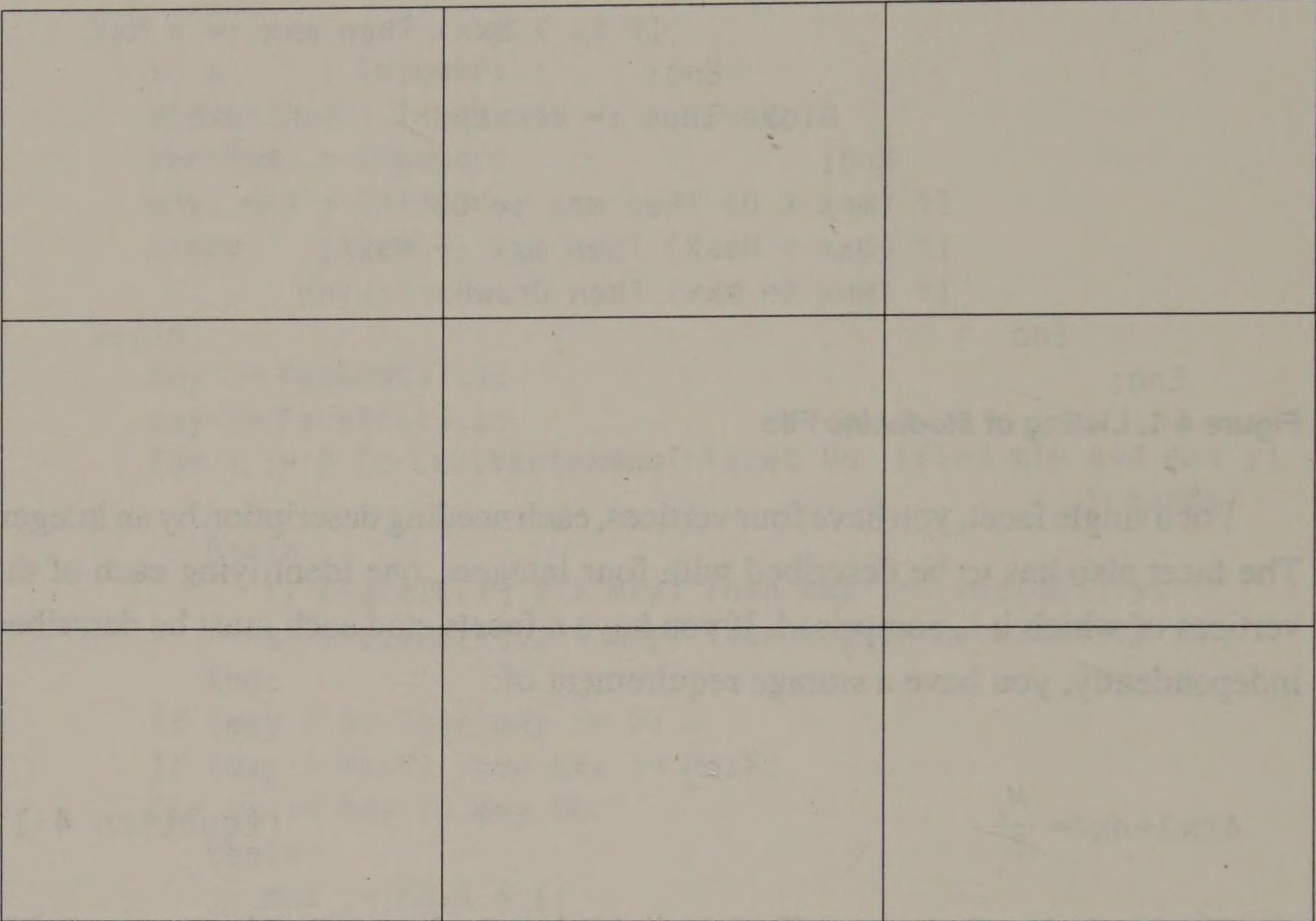


Figure 4-2. Surface Composed of 9 Facets and 16 Vertices

Loading and Saving *Vertex* and *Facet* Data

The next routines considered in the listing are those involved with loading and saving the facet data. The first of these procedures, *InitVertexBuffer*, initializes the various buffers. First, this procedure computes two inverse scale parameters and fills all of the vectors in the *Vertex* array with null vectors. All of the members of the *Facet* array are filled with zeroes.

The next chapter, shows how to add objects to the object list database. Each object will have its own file. That chapter, goes into more detail on the structure of the file. In the paragraphs that follow, assume that an object file already exists and load it back into the working program, or alternately, that you have all the necessary data stored in the program and want to save it to a disk file.

The name of an object data file passes to the procedure *LoadData*. It begins by opening the file of this name and reading the last vertex, last facet and last vertex in facet numbers from it. For each vertex, it reads the three pairs of bytes that make up the x, y, and z coordinates of the vertex. After reading each pair of bytes, it converts them into an integer format. (Please observe that we are using a text type file, but that in the interests of conserving space, we want vertex and facet data to be in the form of integers stored in two bytes, instead of as a string of numbers.)

In the next paragraph you'll see how integers are stored in this manner. If you look at the listing of the *LoadData* procedure, you'll see how to convert the two stored bytes back to an integer again. After reading each set of coordinates for a vertex, the procedure checks if they are within the range of values allowed for such data. If not, the program displays an error message and halts. If the coordinates are within range, they are converted to vector form and stored in the proper member of the *Vertex* array. For each facet, the procedure reads four pairs of bytes. Each pair indicates the number of a vertex that is part of the facet. Each pair converts to an integer and is stored in the proper member of the *Facet* array. After all facets are loaded, the file is closed and the procedure is complete.

The *SaveData* procedure works in an opposite manner from the *LoadData* procedure described above. An object data file name passes to it and opens as a text file for writing. The last vertex and last facet, and the last vertex in facet numbers are written to the file. The procedure begins a *For* loop which iterates for each vertex. At each iteration, a vertex vector converts to three integer coordinates and each coordinate is stored in the form of two bytes. The first byte consists of the eight most significant bits of the integer (the most significant byte) shifted one byte to the right so as to be a character, and this character is written to the file. For the second byte, the integer is anded with 255, leaving only the least significant byte, which is written to the file as the second character. When this loop is complete, a similar process writes out the four vertex numbers for every facet in the two-byte form using the same conversion technique. The file closes and the procedure is complete.

Drawing a Wire Frame Facet

The *PutWireFacet* procedure actually draws the outline of the facet to the screen. It is passed an array of pairs of points, which are the two-dimensional screen coordinates of the four vertices of a facet, and a color value. It runs a *For* loop which draws a line between each adjacent pair of points to frame the facet.

Painting a Solid Facet on the Screen

The procedure *PutFacet* fills in a facet on the screen with the designated color and intensity. It begins by looping to determine the minimum and maximum values of the *y* coordinate for the facet. A *For* loop iterates once for every value of *y* within the facet. Within the loop, the procedure computes the slope of the lines connecting critical pairs of vertices and uses the slopes to determine the beginning and ending values of *x* for the current value of *y*. These values are tested to ensure that they never come out smaller than the minimum *x* for the facet, nor larger than the maximum *x*. If the maximum is greater than or equal to the minimum, the procedure *DrawHorizLine* is called to draw a line of the proper color for that *y* value. This procedure simply uses *PutPixel* to plot points at the desired *y* value for all values of *x* from the beginning to end value. The loop continues until all values of *y* within the facet have had their associated lines drawn, resulting in the facet being filled with the desired shade of color. You may have wondered why the Turbo Pascal procedure *FillPoly* wasn't used to fill the facet, thereby avoiding all of the additional programming. The answer is that *FillPoly* can only fill with 16 colors, whereas you'll need to be able to select from 256 colors for this application. If you had source code available for *FillPoly* it would probably be a fairly simple modification to adapt it for 256 colors; since you do not, you'll need to create it for your own procedure.

A Manually Generated Data File Sample

In Chapter 9, you'll learn how to generate some object data files with computer assistance. But first, let's see what a simple object file might look like. Create a file for an *xy* plane filling the entire display screen. The first line of this file is

4 1 4

SOLID MODELING THEORY AND DATABASE STRUCTURE

indicating that there are four vertices, one facet, and that the facet has four vertices. Next come four sets of three coordinates, giving the coordinates of each of the four vertices. Since the plane is in the xy system, all values of z are zero. The other two coordinates for each vertex are at the maximum limits permitted. Thus you have

23000	23000	0
-23000	23000	0
-23000	-23000	0
23000	-23000	0

Finally, you have a line that lists the vertex numbers for each of the four vertices that comprise the facet. They are:

1	2	3	4
---	---	---	---

Adding Objects to the Scene

In this chapter you will add objects to a scene. Assume that each type of object (cube, sphere, cone, etc.) you are interested in will have an appropriately named disk file containing the necessary vertex and facet information for creating a generic object of the desired type. An array called *ObjList* is needed to provide the necessary information to create a scene which contains a record of information on each object to be displayed.

The first item in this array is *ObjectName*, the name of the object file for the desired object type (for example *Sphere.Dat*). Next are *Tx*, *Ty*, and *Tz*, which are the three-dimensional coordinates of the translation vector. This vector moves the object from its nominal position at the center of the screen to the desired location. Next are *Rx*, *Ry*, and *Rz*, which are the three-dimensional coordinates of the rotation vector. This vector rotates the object from its nominal orientation to a new orientation specified by this vector. Next are *Sx*, *Sy*, and *Sz*, which are the three-dimensional coordinates of the scaling vector. This vector changes the object from its nominal size by scaling its size in each coordinate as specified by the vector. Note that the scale factor for an object can be changed separately in each coordinate direction. This makes it possible, for example, to change a sphere to an ellipsoid, or a cube to a rectangular solid. The next parameter is *Reflection*, which is a Boolean *True* or *False* indicating if the result of this parameter is a reflection or an actual object. Next is *Color*, a byte that indicates the color of the object, then *Sortable*, another Boolean variable which indicates if the object is to be a part of the list of objects to be sorted before display. Finally, a Boolean variable, *Mirror*, indicates if the object is itself a mirror (reflecting object).

The definition of the array for listing objects and the associated procedures are included in the file *AddObjs.Inc*, which is listed in Figure 5-1. The procedures are described in the sections that follow.

Initializing the Object Buffer

The procedure *InitObjectBuffer* initializes the list of objects described above. It begins by setting *ObjectName* as a NULL string. Then the components of the translation, rotation, and scaling vectors are all set to zeroes. *Reflection* is set to *False*, *Color* is set to zero, *Sortable* is set to *False*, and *Mirror* is set to *False*. This procedure is repeated for the 40 objects, which is the maximum allowable number of objects.

Inserting Object Information into the Object Buffer

Here is a brief description of procedures that fill an object record with information. The record number is given by *ObjectNum*. The procedure *AddObject* is passed a string (the name) of the generic object data file for the particular object desired. The procedure writes this string to the proper item in the designated object record. The procedure *Scale* is passed a set of three scaling coordinates, one for each dimension. It writes these coordinates to the proper items in the designated object record. The procedure *Rotate* is passed a set of three rotation coordinates, one for each dimension. It writes these coordinates to the proper items in the designated object record. The procedure *Translate* is passed a set of three translation coordinates, one for each dimension. It writes these coordinates to the proper items in the designated object record. The procedure *ReflectObject* is passed a Boolean value to establish if the object is a reflection. It writes this value to the proper item in the designated object record. The procedure *ObjectColor* is passed a byte which represents the object color. It writes this byte to the proper item in the designated object record. The procedure *AllowSort* is passed a Boolean value to establish if the object is to be a part of the list of objects to be sorted. It writes this value to the proper item in the designated object record. The procedure *Mirrored* is passed a Boolean value to establish if the object is a mirror. It writes this value to the proper item in the designated object record.

ADDING OBJECTS TO THE SCENE

{

Add Objects to the Object List Database

InitObjectBuffer - initialize object buffer
AddObject - add object of certain name
Scale - scale the object
Rotate - rotate the object
Translate - translate the object
ReflectObject - will this object reflect on a reflective object

ObjectColor - color of object
AllowSort - is this object sorted
Mirrored - is the object reflective

}

Const

NumObjects = 40;

Var

ObjectNum : Integer;

LastObject : Integer;

ObjList : Array[1..NumObjects] Of Record

ObjectName : Name;

Tx, Ty, Tz : Real;

Rx, Ry, Rz : Real;

Sx, Sy, Sz : Real;

Reflection : Boolean;

Color : Byte;

Sortable : Boolean;

Mirror : Boolean

End;

Procedure InitObjectBuffer;

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
Begin
  For ObjectNum := 1 To NumObjects Do
    With ObjList[ObjectNum] Do
      Begin
        ObjectName := '';
        Tx := 0;
        Ty := 0;
        Tz := 0;
        Rx := 0;
        Ry := 0;
        Rz := 0;
        Sx := 0;
        Sy := 0;
        Sz := 0;
        Reflection := False;
        Color := 0;
        Sortable := False;
        Mirror := False
      End;
      ObjectNum := 0
    End;
```

```
Procedure AddObject(FileName : Name);
```

```
Begin
  ObjectNum := ObjectNum + 1;
  With ObjList[ObjectNum] Do
    ObjectName := FileName;
    LastObject := ObjectNum
  End;
```

```
Procedure Scale( x, y, z : Real );
```

```
Begin
  With ObjList[ObjectNum] Do
    Begin
      Sx := x;
      Sy := y;
```


ADDING OBJECTS TO THE SCENE

```
        Sz := z  
    End  
End;
```

```
Procedure Rotate( x, y, z : Real );
```

```
    Begin  
        With ObjList[ObjectNum] Do  
            Begin  
                Rx := -x;  
                Ry := -y;  
                Rz := -z  
            End  
        End  
    End;
```

```
Procedure Translate( x, y, z : Real );
```

```
    Begin  
        With ObjList[ObjectNum] Do  
            Begin  
                Tx := -x;  
                Ty := -y;  
                Tz := -z  
            End  
        End  
    End;
```

```
Procedure ReflectObject( State : Boolean );
```

```
    Begin  
        With ObjList[ObjectNum] Do  
            Reflection := State  
        End  
    End;
```

```
Procedure ObjectColor( Col : Byte );
```

```
    Begin
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
    With ObjList[ObjectNum] Do
        Color := Col
    End;
```

```
Procedure AllowSort( State : Boolean );
```

```
    Begin
        With ObjList[ObjectNum] Do
            Sortable := State
        End;
```

```
Procedure Mirrored( State : Boolean );
```

```
    Begin
        With ObjList[ObjectNum] Do
            Mirror := State
        End;
```

```
{
```

Routines to Add Edge Reflectors to a Scene
--

```
    AddEdgeReflectorsAtZero - adds edge reflectors at x=-100,
                                y=-100 and z=0
    AddEdgeReflectors      - adds edge reflectors at x=-100, y=-100
                                and z=-100
```

```
}
```

```
Var
    EdgeReflectorAtZero : Boolean;
    EdgeReflector       : Boolean;
```

```
Procedure AddEdgeReflectorsAtZero;
```


ADDING OBJECTS TO THE SCENE

Begin

EdgeReflectorAtZero := True;

AddObject('Grid.Dat');

Scale (100.0, 100.0, 100.0); { Sx, Sy, Sz }

Rotate (0.0, 0.0, 0.0); { Rx, Ry, Rz }

Translate(0.0, 0.0, 0.0); { Tx, Ty, Tz }

ReflectObject(False); { bottom at z=0 }

ObjectColor(7);

AllowSort(False);

Mirrored(True);

AddObject('Grid.Dat');

Scale (50.0, 100.0, 50.0); { Sx, Sy, Sz }

Rotate (0.0, 90.0, 0.0); { Rx, Ry, Rz }

Translate(-100.0, 0.0, 50.0); { Tx, Ty, Tz }

ReflectObject(False); { right back at x=-100 }

ObjectColor(7);

AllowSort(False);

Mirrored(True);

AddObject('Grid.Dat');

Scale (100.0, 50.0, 50.0); { Sx, Sy, Sz }

Rotate (-90.0, 0.0, 0.0); { Rx, Ry, Rz }

Translate(0.0, -100.0, 50.0); { Tx, Ty, Tz }

ReflectObject(False); { left back at y=-100 }

ObjectColor(7);

AllowSort(False);

Mirrored(True)

End;

Procedure AddEdgeReflectors;

Begin

EdgeReflector := True;

AddObject('Grid.Dat');

Scale (100.0, 100.0, 100.0); { Sx, Sy, Sz }

Rotate (0.0, 0.0, 0.0); { Rx, Ry, Rz }

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
Translate( 0.0, 0.0, -100.0 ); { Tx, Ty, Tz }
ReflectObject(False); { bottom at z=-100 }
ObjectColor(7);
AllowSort(False);
Mirrored(True);

AddObject('Grid.Dat');
Scale ( 100.0, 100.0, 100.0 ); { Sx, Sy, Sz }
Rotate ( 0.0, 90.0, 0.0 ); { Rx, Ry, Rz }
Translate( -100.0, 0.0, 0.0 ); { Tx, Ty, Tz }
ReflectObject(False); { right back at x=-100 }
ObjectColor(7);
AllowSort(False);
Mirrored(True);

AddObject('Grid.Dat');
Scale ( 100.0, 100.0, 100.0 ); { Sx, Sy, Sz }
Rotate ( -90.0, 0.0, 0.0 ); { Rx, Ry, Rz }
Translate( 0.0, -100.0, 0.0 ); { Tx, Ty, Tz }
ReflectObject(False); { left back at y=-100 }
ObjectColor(7);
AllowSort(False);
Mirrored(True)
End;

{
```

Add Objects to Scene From Procedure

AddObjectsToSceneFromProcedure - add object to scene from a procedure to allow calculation of translation, scaling and rotation

```
}

Procedure AddObjectsToSceneFromProcedure;
```

```
{
```


ADDING OBJECTS TO THE SCENE

Stacked Sphere Scene Generator

```
}
```

```
Procedure StackedSpheres;
```

```
Var
```

```
  r, s : Real;
```

```
Begin
```

```
  r:= 16.0; { diameter of sphere (r=12.0) }  
  s:= 16.0; { displacement of sphere from z=0 plane }
```

```
  AddObject('Sphere.Dat');
```

```
  Scale (      r,      r,      r );    { Sx, Sy, Sz }
```

```
  Rotate  (  0.0,  0.0,  0.0 );    { Rx, Ry, Rz }
```

```
  Translate( -2*r,  -r,  s+r );    { Tx, Ty, Tz }
```

```
  ReflectObject(True);
```

```
  ObjectColor(1);
```

```
  AllowSort(True);
```

```
  Mirrored(False);
```

```
  AddObject('Sphere.Dat');
```

```
  Scale (      r,      r,      r );    { Sx, Sy, Sz }
```

```
  Rotate  (  0.0,  0.0,  0.0 );    { Rx, Ry, Rz }
```

```
  Translate(  0.0,  -r,  s+r );    { Tx, Ty, Tz }
```

```
  ReflectObject(True);
```

```
  ObjectColor(2);
```

```
  AllowSort(True);
```

```
  Mirrored(False);
```

```
  AddObject('Sphere.Dat');
```

```
  Scale (      r,      r,      r );    { Sx, Sy, Sz }
```

```
  Rotate  (  0.0,  0.0,  0.0 );    { Rx, Ry, Rz }
```

```
  Translate(  2*r,  -r,  s+r );    { Tx, Ty, Tz }
```

```
  ReflectObject(True);
```

```
  ObjectColor(3);
```

```
  AllowSort(True);
```

```
  Mirrored(False);
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
AddObject('Sphere.Dat');
Scale (    r,    r,    r );    { Sx, Sy, Sz }
Rotate  (  0.0,  0.0,  0.0 );    { Rx, Ry, Rz }
Translate(    -r, 0.866*r,  s+r );    { Tx, Ty, Tz }
ReflectObject(True);
ObjectColor(4);
AllowSort(True);
Mirrored(False);
```

```
AddObject('Sphere.Dat');
Scale (    r,    r,    r );    { Sx, Sy, Sz }
Rotate  (  0.0,  0.0,  0.0 );    { Rx, Ry, Rz }
Translate(    r, 0.866*r, s+r );    { Tx, Ty, Tz }
ReflectObject(True);
ObjectColor(5);
AllowSort(True);
Mirrored(False);
```

```
AddObject('Sphere.Dat');
Scale (    r,    r,    r );    { Sx, Sy, Sz }
Rotate  (  0.0,  0.0,  0.0 );    { Rx, Ry, Rz }
Translate(  0.0, 2.732*r,  s+r );    { Tx, Ty, Tz }
ReflectObject(True);
ObjectColor(6);
AllowSort(True);
Mirrored(False);
```

```
AddObject('Sphere.Dat');
Scale (    r,    r,    r );    { Sx, Sy, Sz }
Rotate  (  0.0,  0.0,  0.0 );    { Rx, Ry, Rz }
Translate(    -r, -0.293*r, s+2.6*r );    { Tx, Ty, Tz }
ReflectObject(True);
ObjectColor(7);
AllowSort(True);
Mirrored(False);
```

```
AddObject('Sphere.Dat');
Scale (    r,    r,    r );    { Sx, Sy, Sz }
Rotate  (  0.0,  0.0,  0.0 );    { Rx, Ry, Rz }
Translate(    r, -0.293*r, s+2.6*r );    { Tx, Ty, Tz }
ReflectObject(True);
```


ADDING OBJECTS TO THE SCENE

```
ObjectColor(1);
AllowSort(True);
Mirrored(False);
```

```
AddObject('Sphere.Dat');
Scale (    r,    r,    r );    { Sx, Sy, Sz }
Rotate  (  0.0,  0.0,  0.0 );    { Rx, Ry, Rz }
Translate( 0.0,  1.5*r, s+2.6*r );    { Tx, Ty, Tz }
ReflectObject(True);
ObjectColor(2);
AllowSort(True);
Mirrored(False);
```

```
AddObject('Sphere.Dat');
Scale (    r,    r,    r );    { Sx, Sy, Sz }
Rotate  (  0.0,  0.0,  0.0 );    { Rx, Ry, Rz }
Translate( 0.0,  0.5*r, s+4.4*r );    { Tx, Ty, Tz }
ReflectObject(True);
ObjectColor(3);
AllowSort(True);
Mirrored(False)
```

```
End;
```

```
Begin
```

```
InitPerspective(True, 0, -25, 450, 500);
InitPlotting(240, 18);
GetViewVector;
InitLightDirection(45, 45);
GetLightVector;
VerticalSort := True;
AddEdgeReflectorsAtZero;
StackedSpheres
```

```
End;
```

```
{
```

Add Objects to Scene from .SCN Disk File

AddObjectsToSceneFromDiskFile - add objects to database from

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

.SCN Disk File

}

Procedure AddObjectsToSceneFromDiskFile(FileName : Name);

Type

Strg = String[5];

Var

B : Strg;

I1, I2, I3, I4 : Integer;

R1, R2, R3 : Real;

ObjectFileName : Name;

Function Bool(B : Strg) : Boolean;

Begin

If (B = 'FALSE') Then

Bool := False

Else

Bool := True

End;

Begin

FileName := FileName + '.Scn';

Assign(TextDiskFile, FileName);

Reset(TextDiskFile);

ReadLn(TextDiskFile); { comment line }

ReadLn(TextDiskFile); { blank }

ReadLn(TextDiskFile, B, I1, I2, I3, I4);

{ perspective, Mx, My, Mz, D }

InitPerspective(Bool(B), I1, I2, I3, I4);

ReadLn(TextDiskFile, I1, I2); { image angle and tilt }

InitPlotting(I1, I2);

ADDING OBJECTS TO THE SCENE

GetViewVector;

ReadLn(TextDiskFile, I1, I2); { light angle and tilt }

InitLightDirection(I1, I2);

GetLightVector;

ReadLn(TextDiskFile, B); { vertical sort }

VertSort(Bool(B));

ReadLn(TextDiskFile, B); { turn edge reflectors on }

If Bool(B) Then AddEdgeReflectors;

ReadLn(TextDiskFile, B); { turn edge reflectors on }

If Bool(B) Then AddEdgeReflectorsAtZero;

Repeat

ReadLn(TextDiskFile); { blank }

ReadLn(TextDiskFile, ObjectFileName); { object name }

AddObject(ObjectFileName);

ReadLn(TextDiskFile, I1); { object color }

ObjectColor(I1);

ReadLn(TextDiskFile, R1, R2, R3); { scale object }

Scale(R1, R2, R3);

ReadLn(TextDiskFile, R1, R2, R3); { rotate object }

Rotate(R1, R2, R3);

ReadLn(TextDiskFile, R1, R2, R3); { translate object }

Translate(R1, R2, R3);

ReadLn(TextDiskFile, B); { object can be reflected }

ReflectObject(Bool(B));

ReadLn(TextDiskFile, B); { object can be sorted }

AllowSort(Bool(B));

ReadLn(TextDiskFile, B); { object is a mirror }


```

        Mirrored(Bool(B))

    Until EOF(TextDiskFile);

    Close(TextDiskFile)
End;

```

Figure 5-1. Listing of the *AddObjs.Inc* File

Adding Edge Reflectors to a Scene

One feature that is often used in scenes is to paint three walls onto the scene, each consisting of a number of small, square, mirrored tiles, separated by a black grid. Depending upon the scene, these walls will have their intersection at $(-100, -100, 0)$, which results in a floor at about the center of the screen and two back walls, or $(-100, -100, -100)$, which results in a floor near the bottom of the screen and two back walls. Since these are used so frequently, two procedures are set up to paint them to the screen.

The first case uses *AddEdgeReflectorsAtZero*. This procedure uses the object data file *Grid.Dat*, which contains the facet and vertex information for a tiled plane. The procedure calls *Grid.Dat* three times, each time with the proper translation, scaling, and rotation to draw one of the walls. For each wall, it sets *ReflectObject* to *False* to indicate that this is an actual object, not a reflection. It sets *ObjectColor* to 7, which results in the tiles being a light gray. It sets *AllowSort* to *False*. Since these walls are the boundaries of the scene, they must be drawn first and therefore should not be sorted. It sets *Mirrored* to *True*, since all of the wall tiles are mirrored and reflect the object that makes up the scene.

The second case uses the procedure *AddEdgeReflectors*. This procedure is exactly like the previous one except that the translations, rotations and scalings are modified to produce the new wall locations.

Adding Objects to a Scene from a Procedure

The procedure *AddObjectsToSceneFromProcedure* may be selected as an option using the *Model* program. It begins by calling the *InitPerspective* and *InitPlotting* procedures, described in Chapter 2. Next it calls *GetViewVector* to establish a unit vector in the direction of the viewer, calls *InitLightDirection* to establish the direction of the light source, and *GetLightVector* to set up a unit vector in the direction of the light source. (These three procedures will be described in Chapter 7.) Next, the procedure *VertSort* is called. This procedure simply sets the parameter *VerticalSort* to *True*, which causes objects to be sorted in the proper order later. Next *AddEdgeReflectorsAtZero* is called, producing the reflective wall and floor tiles as described previously. Finally, the procedure *StackedSpheres* is called to produce a display of stacked spheres. This procedure begins by setting a diameter for a sphere and its displacement from the $z=0$ plane. The *AddObject* procedure is called to add this sphere to the data base with the color blue. The procedure is called again to add a green sphere to the database with a different set of translation coordinates. This process continues until ten different spheres have been added to the data base, using seven different colors and with differing translation coordinates, adjusted so that the spheres form a pyramidal stack.

If you want to use *AddObjectsToSceneFromProcedure* to generate your own scene, you may use it with the predetermined mirrored walls, viewer position, and light source by changing the name of the procedure called in the last statement from *StackedSpheres* to your own procedure name. Within your new procedure, include a number of calls to *AddObject* using the name of one of the generic object data files. Then provide calls for each object to scale the size of the object, to rotate it, to translate it to the position you want, to specify whether you want its reflections to be displayed, to identify its color, to state whether it is to be sorted, and to determine whether it is to be a mirror or not. If you are not satisfied with the default values for the mirrored walls, viewer and light source positions, you will have to modify the default values that are part of the procedure *AddObjectsToSceneFromProcedure* in addition to creating your own procedure full of scene data.

Adding Objects to a Scene From a Disk File

In Chapter 10, is a description of how to create a *.SCN* file containing the data required for a scene. Here, it is assumed that you already have such a file and want to transfer it to the variables and arrays from which the scene will be created. The procedure *AddObjectsToSceneFromDiskFile* performs this task.

This procedure is passed a string which is the file name of the scene data, without the extension. The first thing that happens is that the extension *.SCN* is appended to this file name. The file is opened and the first two lines are read: The first is a comment line and the second is a blank line. Nothing is done with these lines. They are for observation when examining the file with a debugger program. Next a line of data is read containing the parameters for the perspective of the scene. These parameters are passed to the *InitPerspective* procedure, which then sets up the perspective for the scene. The next line read from the file contains the image and tilt angles. The procedure *InitPlotting* is then passed these parameters and sets up the viewer position from them. The procedure *GetViewVector* is called to establish a unit vector in the position of the viewer. The next line read from the file contains light-angle information. This is passed to the procedure *InitLightDirection* to set up the light direction. The procedure *GetLightVector* is called to set up a unit vector in the light direction. The next line read determines if *VerticalSort* shall be turned on. The parameter is set to the Boolean value read. The next two lines determine if the edge reflectors are to be generated. If the first line is *True*, *AddEdgeReflectors* is called. If the second line is *True*, *AddEdgeReflectorsAtZero* is called. A *Repeat* loop continues until the end of the file is reached. At each iteration of this loop, the procedure begins by reading a blank line. It reads the file name of the object data file. This is passed to the proper parameter by *AddObject*. Then lines are read from the disk file for the object color, the scaling factors, the rotation parameters, the translation parameters, if the object can be reflected, if the object can be sorted, and if the object is a mirror. Each of these sets of information is loaded into its proper parameters by the associated procedure. Once the entire file has been read, it is closed and the procedure is complete.

Sorting and Displaying Objects on the Screen

This chapter is concerned with two subjects. The first is the sorting of objects to process them in proper order. One of the problems with having a scene containing several objects at different distances is that the closer objects must overlap the farther ones (as they would appear in a photograph or painting). This is solved by simply drawing the farthest object first and the others in the order of their distance from the observer. The closest object is drawn right over the others and the appearance of the picture is correct. In order to do this, you have to sort the list of objects so the objects are in proper order with the farthest first and the closest last. Likewise, sort the lists for reflections of objects in the x , y , and z planes. The second subject to be considered is the actual display of objects on the screen. This involves both the actual objects and their reflections on the three planes. Displaying an object requires converting each facet to two-dimensional screen coordinates and painting the resulting facet on the screen in the shade of color determined by the color of the object, the direction of the light source and observer, and the surface normal vector of the facet. The program listing for both these subjects is in the file *DispObjs.Inc.*, listed in Figure 6-1.

Sorting Objects for Placement on the Screen

Look at the procedure *PrepareSort* which prepares the sorting process. This procedure begins by checking to see if there is only one object in the object list. If this is the case, there are no mirrors and you won't need to worry about either sorting or reflections. If there is more than one object, the procedure begins a *While* loop which iterates until an object in the object list is tagged as sortable. If such an object is found,

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

the parameter *First* is set to the number of this first, sortable object. Another *While* loop begins which checks for the existence of objects with a *Mirror* tag. If there are mirrored objects, exit this loop with the parameter *ReflSurface* set to *True*, indicating that mirrored objects exist, and the parameter *FirstObject* set to the number in the list of the first, non-mirrored object. The number of mirrors, *NumMirrors*, is set to one less than this number. Next, the procedure checks to see if the first sortable object is the last object. If so, there is no need to sort, so *Sorting* is set to *False*. Otherwise, *Sorting* is set to *True*. This completes this procedure.

```
{
```

Routines for Sorting Objects for Order of Placement

PrepareSort	- setup for sorting routines
SortObjectsBackToFront	- sort for actual objects
SortTopToBottom	- sort for reflections in Z
SortBottomToTop	- sort for actual objects
SortObjectsFrontToBackInY	- sort for reflections in Y
SortObjectsFrontToBackInX	- sort for reflections in X

```
}
```

```
Var
```

```
    First      : Integer;  
    Last       : Integer;  
    FirstObject : Integer;  
    Sorting    : Boolean;  
    ReflSurface : Boolean;  
    NumMirrors : Integer;
```

```
Procedure PrepareSort;
```

```
Begin
```

```
    ReflSurface := False;
```

```
    Last := LastObject;
```

```
    If (LastObject = 1) Then { if only one object }
```

```
    Begin
```

```
        First := Last; { then no sorting }
```


SORTING AND DISPLAYING OBJECTS ON THE SCREEN

```
    FirstObject := LastObject { and no mirrors }
End
Else
Begin
    First := 1; { find first sortable object }
    While (ObjList[First].Sortable = False) And (First < Last)
        Do
            First := First + 1;
    FirstObject := 1; { find first non-mirrored object }
    While (ObjList[FirstObject].Mirror = True) And
        (FirstObject < LastObject) Do
        Begin
            FirstObject := FirstObject + 1;
            ReflSurface := True { we have at least 1 reflective
                                surface }
        End
    End;
    NumMirrors := FirstObject - 1;
    If (First = Last) Then { if the first sortable object is the
                            last }
        Sorting := False {object then there is no need to sort }
    Else
        Sorting := True
End;

Var
    Index : Integer;

Procedure SwapData;

Procedure SwapStrings(Var A, B : Name);

Var
    T : Name;

Begin
    T := A;
    A := B;
    B := T
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
End;

Procedure SwapRealNum(Var A, B : Real);

Var
    T : Real;

Begin
    T := A;
    A := B;
    B := T
End;

Procedure SwapByteNum(Var A, B : Byte);

Var
    T : Byte;

Begin
    T := A;
    A := B;
    B := T
End;

Procedure SwapBoolean(Var A, B : Boolean);

Var
    T : Boolean;

Begin
    T := A;
    A := B;
    B := T
End;

Begin
    SwapStrings( ObjList[Index].ObjectName, ObjList[Index +
                                                         1].ObjectName);
    SwapRealNum( ObjList[Index].Tx, ObjList[Index+1].Tx );
    SwapRealNum( ObjList[Index].Ty, ObjList[Index+1].Ty );
```


SORTING AND DISPLAYING OBJECTS ON THE SCREEN

```
SwapRealNum( ObjList[Index].Tz, ObjList[Index+1].Tz );
SwapRealNum( ObjList[Index].Rx, ObjList[Index+1].Rx );
SwapRealNum( ObjList[Index].Ry, ObjList[Index+1].Ry );
SwapRealNum( ObjList[Index].Rz, ObjList[Index+1].Rz );
SwapRealNum( ObjList[Index].Sx, ObjList[Index+1].Sx );
SwapRealNum( ObjList[Index].Sy, ObjList[Index+1].Sy );
SwapRealNum( ObjList[Index].Sz, ObjList[Index+1].Sz );
SwapBoolean( ObjList[Index].Reflection, ObjList[Index +
                                                    1].Reflection );
SwapByteNum( ObjList[Index].Color, ObjList[Index +
                                                    1].Color );
SwapBoolean( ObjList[Index].Sortable, ObjList[Index +
                                                    1].Sortable );
SwapBoolean( ObjList[Index].Mirror, ObjList[Index +
                                                    1].Mirror )

End;

Var
  I : Integer;

Procedure SortObjectsBackToFront;
  Procedure CheckForSwap(v, A, B : Real);

  Begin
    If v >= 0 Then
      Begin
        If ( A < B ) Then
          SwapData
        End
      Else
        If ( A > B ) Then
          SwapData
        End
      End;

  Procedure OrderX;

  Begin
    For I := First To Last - 1 Do
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
    For Index := First To Last - 1 Do
        CheckForSwap(View[0], ObjList[Index].Tx,
            ObjList[Index + 1].Tx)
    End;

    Procedure OrderY;

    Begin
        For I := First To Last - 1 Do
            For Index := First To Last - 1 Do
                CheckForSwap(View[1], ObjList[Index].Ty,
                    ObjList[Index + 1].Ty)
            End;
        End;

    Procedure OrderZ;

    Begin
        For I := First To Last - 1 Do
            For Index := First To Last - 1 Do
                CheckForSwap(View[2], ObjList[Index].Tz,
                    ObjList[Index + 1].Tz)
            End;
        End;

    Var
        x, y, z : Real;

    Begin
        x := Abs(View[0]);
        y := Abs(View[1]);
        z := Abs(View[2]);
        If (x > y) And (x > z) Then OrderX
        Else
            If (y > x) And (y > z) Then OrderY
            Else
                OrderZ
        End;

    End;

    Procedure SortTopToBottom;
```


SORTING AND DISPLAYING OBJECTS ON THE SCREEN

```
Procedure SwapDown(A, B : Real);
```

```
Begin
  If ( A > B ) Then
    SwapData
End;
```

```
Begin
  For I := First To Last - 1 Do
    For Index := First To Last - 1 Do
      SwapDown(ObjList[Index].Tz, ObjList[Index + 1].Tz)
    End;
  End;
```

```
Procedure SortBottomToTop;
```

```
Procedure SwapUp(A, B : Real);
```

```
Begin
  If ( A < B ) Then
    SwapData
  End;
```

```
Begin
  For I := First To Last - 1 Do
    For Index := First To Last - 1 Do
      SwapUp(ObjList[Index].Tz, ObjList[Index + 1].Tz)
    End;
  End;
```

```
Procedure SortObjectsFrontToBackInY;
```

```
Procedure CheckForSwap(v, A, B : Real);
```

```
Begin
  If v >= 0 Then
    Begin
      If ( A > B ) Then
        SwapData
      End
    End
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
Else
    If ( A < B ) Then
        SwapData
End;

Procedure OrderY;

Begin
    For I := First To Last - 1 Do
        For Index := First To Last - 1 Do
            CheckForSwap(View[1], ObjList[Index].Ty,
                          ObjList[Index + 1].Ty)
        End;
    End;

Begin
    OrderY
End;

Procedure SortObjectsFrontToBackInX;

Procedure CheckForSwap(v, A, B : Real);

Begin
    If v >= 0 Then
        Begin
            If ( A > B ) Then
                SwapData
            End
        End
    Else
        If ( A < B ) Then
            SwapData
        End;
    End;

Procedure OrderX;
Begin
    For I := First To Last - 1 Do
        For Index := First To Last - 1 Do
            CheckForSwap(View[0], ObjList[Index].Tx,
                          ObjList[Index + 1].Tx)
```


SORTING AND DISPLAYING OBJECTS ON THE SCREEN

End;

Begin

OrderX

End;

{

Place an Object onto the Screen

PlaceObjectOnScreen - place an object onto the screen in Vertex Format, Wire Frame Format or Solid Model Format

}

Var

XReflectedObject : Boolean;

YReflectedObject : Boolean;

ZReflectedObject : Boolean;

Procedure PlaceObjectOnScreen(Color : Byte);

Procedure PreviewVertices;

Var

x, y, z : Real;

Pt : TDA;

Begin

For FacetNum := 1 To LastFacet Do

For VertexNumInFacet := 1 To LastVertexNumInFacet Do

Begin

VertexNum := Facet[FacetNum, VertexNumInFacet];

VecScaleMultI(InvScaleImage, Vertex[VertexNum], Pt);

UnVec(Pt, x, y, z);

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
        CartesianPlot3D(x, y, z, (MaxInten + 1) * (Color -
                                1) + MaxInten)
    End
End;

Procedure WireFrameDiagram;

Var
    Face3d : VoxelArray;
    Face2d : PixelArray;

Begin
    For FacetNum := 1 To LastFacet Do
    Begin
        For VertexNumInFacet := 1 To LastVertexNumInFacet Do
        Begin
            VertexNum := Facet[FacetNum, VertexNumInFacet];
            VecScaleMultI(InvScaleImage, Vertex[VertexNum],
                          Face3d[VertexNumInFacet])
        End;
        If Visible(Face3d) Then
        Begin
            GetProjectedCoords(Face3d, Face2d);
            PutFacet(Face2d, 1, Black);
            PutWireFacet(Face2d, (MaxInten + 1) * (Color - 1)
                          + MaxInten)
        End
    End
End;

Procedure SolidModelDiagram;

Var
    Intens : Byte;
    Face3d : VoxelArray;
    Face2d : PixelArray;

Begin
    MirrorX := False;
    MirrorY := False;
```


SORTING AND DISPLAYING OBJECTS ON THE SCREEN

```
MirrorZ := False;
If ZReflectedObject Then
    MirrorZ := True
Else
    If YReflectedObject Then
        MirrorY := True
    Else
        If XReflectedObject Then
            MirrorX := True;
For FacetNum := 1 To LastFacet Do
Begin
    For VertexNumInFacet := 1 To LastVertexNumInFacet Do
    Begin
        VertexNum := Facet[FacetNum, VertexNumInFacet];
        VecScalMultI(InvScaleImage, Vertex[VertexNum],
                    Face3d[VertexNumInFacet])
    End;
    If Visible(Face3d) Then
    Begin
        Intens := Intensity;
        If (Intens > 0) Then
        Begin
            GetProjectedCoords(Face3d, Face2d);
            PutFacet2(Face2d, Color, Intens)
        End
    End;
End;
End;

Begin
    If Preview Then
        PreviewVertices;
    If WireFrame Then
        WireFrameDiagram;
    If SolidModel And Not(Preview Or WireFrame) Then
        SolidModelDiagram
End;
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
{
```

Display Objects in Scene

DisplayObjectsInScene - displays the objects and any reflections in scene

```
}
```

```
Procedure DisplayObjectsInScene;
```

```
Var
```

```
  LastObjectName : Name;
```

```
  ReflectedObjects : Boolean;
```

```
  Procedure DisplayReflectiveSurfaces;
```

```
  Begin
```

```
    LastObjectName := '';
```

```
    ReflectedObjects := False;
```

```
    Reflect := False; { this itself is not a reflection }
```

```
    For ObjectNum := 1 To FirstObject - 1 Do { reflective  
                                              objects first }
```

```
    With ObjList[ObjectNum] Do
```

```
      Begin { avoid loading repeating objects }
```

```
        If (ObjectName <> LastObjectName) Then
```

```
          Begin
```

```
            InitVertexBuffer;
```

```
            LoadData(ObjectName)
```

```
          End;
```

```
          If Mirror Then
```

```
            Begin
```

```
              ReflectedObjects := True;
```

```
              Mirroring := True; { this is a mirror }
```

```
              AffineTransformation(Tx, Ty, Tz, Sx, Sy, Sz, Rx,  
                                   Ry, Rz);
```

```
              PlaceObjectOnScreen(Color);
```

```
              InvAffineTransformation(Tx, Ty, Tz, Sx, Sy, Sz,  
                                       Rx, Ry, Rz)
```


SORTING AND DISPLAYING OBJECTS ON THE SCREEN

```
End
Else
    Mirroring := False;
    LastObjectName := ObjectName
End;
End;

Procedure DisplayReflections;

Begin
    For ObjectNum := FirstObject To LastObject Do
        With ObjList[ObjectNum] Do
            Begin { avoid loading repeating objects }
                If (ObjectName <> LastObjectName) Then
                    Begin
                        InitVertexBuffer;
                        LoadData(ObjectName)
                    End;
                    If Reflection And Not(Preview Or WireFrame) Then
                        Begin
                            Mirroring := Mirror;
                            Reflect := True;
                            AffineTransformation(Tx, Ty, Tz, Sx, Sy, Sz,
                                                    Rx, Ry, Rz);

                            PlaceObjectOnScreen(Color);
                            InvAffineTransformation(Tx, Ty, Tz, Sx, Sy, Sz,
                                                    Rx, Ry, Rz)
                        End
                    End
                End;
                Reflect := False;
                LastObjectName := ObjectName
            End;
        End;
    End;

Procedure DisplayActualImages;

Begin
    For ObjectNum := FirstObject To LastObject Do
        With ObjList[ObjectNum] Do
            Begin
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
    If (ObjectName <> LastObjectName) Then
    Begin
        InitVertexBuffer;
        LoadData(ObjectName)
    End;
    If Not( (ObjectNum = 1) And Mirror ) Then
    Begin
        Mirroring := Mirror;
        Reflect := False;
        AffineTransformation(Tx, Ty, Tz, Sx, Sy, Sz, Rx,
                               Ry, Rz);
        PlaceObjectOnScreen(Color);
        InvAffineTransformation(Tx, Ty, Tz, Sx, Sy, Sz,
                                Rx, Ry, Rz)
    End;
    LastObjectName := ObjectName
End;

End;

Procedure DefineReflection(XRef, YRef, ZRef : Boolean);

Begin
    XReflectedObject := XRef;
    YReflectedObject := YRef;
    ZReflectedObject := ZRef
End;

Begin
    PrepareSort;
    DefineReflection(False, False, False); { reflective surfaces}
    If Sorting Then SortObjectsBackToFront;
    DisplayReflectiveSurfaces;
    If Not(Preview Or WireFrame) And ReflSurface Then
    Begin
        DefineReflection(False, False, True); {z reflections }
        If Sorting And VerticalSort Then SortTopToBottom;
        If ReflectedObjects Then DisplayReflections;
        If (NumMirrors > 1) Then { do not do x and y reflections }
        Begin
            { if there is only 1 mirror }
            DefineReflection(False, True, False);
            { y reflections }
```


SORTING AND DISPLAYING OBJECTS ON THE SCREEN

```
    If Sorting Then SortObjectsFrontToBackInY;
    If Sorting And VerticalSort Then SortBottomToTop;
    If ReflectedObjects Then DisplayReflections;
    DefineReflection(True, False, False);
        { x reflections }
    If Sorting Then SortObjectsFrontToBackInX;
    If Sorting And VerticalSort Then SortBottomToTop;
    If ReflectedObjects Then DisplayReflections;
End
End;

DefineReflection(False, False, False); { actual objects }
If Sorting Then SortObjectsBackToFront;
If Sorting And VerticalSort Then SortBottomToTop;
DisplayActualImages
End;
```

Figure 6-1. Listing of *DispObjs.Inc* File

The principal tool used in sorting the list of objects is the procedure *SwapData*. This procedure uses individual procedures to swap the position of two, adjacent object descriptions. Note that for each data item, there is a separate *Swap* procedure. Each one is the same in that it places the first item passed to it into temporary storage, moves the second item to the first item position and moves the first item from temporary storage to the second item position. The only difference between these procedures is the data type passed to and from the procedure.

There are several different sorting procedures and which one you use depends on the sorting of actual objects or their reflections in one of the three, possible, reflective planes. *SortObjectsBackToFront* sorts actual objects. This procedure checks the values of the coordinates that make up the viewer's position. The larger of these determines the coordinate of the viewer and object positions to order the object list. A pair of *If* statements call one of the procedures *OrderX*, *OrderY*, or *OrderZ*. The *OrderX* procedure uses a pair of nested *For* loops to perform the pair swaps necessary to completely order the list. For each one, it calls the procedure *CheckForSwap*. If the viewer *x* coordinate is greater than or equal to zero, *CheckSwap* calls *SwapData* to swap the pair if the second is greater than the first. If the viewer *x* coordinate is less than zero, the procedure is reversed; the swap occurs if the second

is less than the first. The *OrderY* and *OrderZ* procedures are just the same except that they test the *y* and *z* coordinates respectively. When the two *For* loops of *SortObjectsBackToFront* are completed, all of the necessary swaps have been made and the object list is in order.

These following procedures are all similar in the way that they perform sorting operations, as seen in the listing. *SortBottomToTop* is used for actual objects, but it simply orders the objects by swapping without reference to the viewer position coordinates. The procedure *SortTopToBottom* is used for sorting reflections in the $z=0$ plane. *SortObjectsFrontToBackInY* sorts reflections in the $y=0$ plane. The procedure *SortObjectsFrontToBackInX* sorts reflections in the $x=0$ plane.

Placing an Object on the Screen

This procedure draws an object on the screen. It actually consists of a call to one, or more, of three procedures. If the parameters *Preview*, *WireFrame*, or *SolidModel* have been set to *True*, the procedures *PreviewVertices*, *WireFrameDiagram*, and *SolidModelDiagram* are called, respectively.

The *PreviewVertices* procedure uses two, nested *For* loops to scan through every facet of the object and every vertex of each facet. Each vertex is scaled for display using the scale factor *InvScaleImage*, defined in Chapter 4. The vertex is converted from a vector to three separate points, used as parameters for the procedure *CartesianPlot3D*, which converts the position to its two-dimensional counterpart and plots a point to the screen. The point is painted in bright yellow.

The procedure *WireFrameDiagram* begins in just the same way as the previous procedure, except instead of plotting each converted vertex to the screen, it is saved in an array *Face3D*. The resulting facet definition is tested to see if that facet is visible. If it is, that facet is converted to the two-dimensional representation with the procedure *GetProjectedCoords* and painted in black on the screen using *PutFacet*. This blanks out any information from facets hidden by the one being drawn. The procedure *PutWireFacet* outlines the facet in bright yellow.

The procedure *SolidModelDiagram* is similar to *WireFrameDiagram*, with a few exceptions. The first difference is that for each facet the procedure sets the parameters *MirrorX*, *MirrorY*, and *MirrorZ*. These parameters indicate if the facets

being processed are reflections in the $x=0$, $y=0$, or $z=0$ planes. If they are, the appropriate mirror variable is set to *True*. If the facets are for actual objects, all three variables are set to false. Everything proceeds just like *WireFrameDiagram* up to the point where the procedure *PutFacet* is called by the wire frame program. In this case, a different version of the procedure, *PutFacet2* is called. This procedure takes into account mirroring, which the *PutFacet* procedure does not. The procedure is called with the actual facet color and intensity, instead of with black. That ends each loop, instead of having an additional call to draw the wire frame.

At this point, you may be confused as to why some of these functions appear in other files and are described there. For example, *PutFacet* is included in *Model.Inc* and is described in Chapter 4, whereas *GetProjectedCoords* and *PutFacet2* are included in *Model.Pas*, described in Chapter 8. The reason is that some of the variables defined in the first procedure are needed before you come to the *DispObjs.Inc* file whereas some of the variables needed for *GetProjectedCoords* and *PutFacet2* are not defined until after *DispObjs.Inc* is used. It is a shame that this makes it harder to track where various procedures are described, but it is essential to the program design.

Displaying Objects and Reflections

The procedure *DisplayObjectsInScene* displays all the objects and reflections that make up the scene. This procedure begins by calling the procedure *PrepareSort* which prepares the sorting process. It calls the procedure *DefineReflectedObject* to indicate that the objects being treated are not reflections. This procedure takes the three parameters passed to it and uses them to install the proper Boolean values in *XReflectedObject*, *YReflectedObject* and *ZReflectedObject*. Next, if the *Sorting* parameter is set, the *SortObjectsBlackToFront* procedure is called to sort the display of actual objects.

The *DisplayReflectiveSurfaces* procedure is then called. This procedure properly sets up parameters and loops through the objects that represent reflective planes. For each one, the vertex buffer is initialized and the object information is loaded. If the *Mirror* parameter is *True* the proper parameters are set, the *AffineTransformation* procedure determines the location of the object's coordinates and

PlaceObjectOnScreen draws the plane to the screen. The *InvAffineTransformation* procedure is called to restore the original parameters. This procedure terminates after it has looped through any or all of the reflective planes in the scene.

Next, the *DisplayObjectsInScene* procedure checks that you are not in the *WireFrame* or *Preview* modes, and the *ReflSurface* parameter is *True*. First, *DefineReflection* defines that the reflection is in the z plane, and the proper sorting for the z plane is performed, if indicated. Then, if the z plane is identified as a reflective surface, the procedure *DisplayReflections* is called. This procedure uses a *For* loop to cycle through all of the objects displayed on the screen. For each object, the vertex buffer is initialized and the data for the object is loaded. If you are not in the *WireFrame* or *Preview* modes, and a reflection is indicated, the proper parameters are set, the affine transformation is performed to determine the location of the object reflection parameters in the $z=0$ plane, and the object is painted on the screen. The inverse affine transformation restores the original parameters and the loop proceeds through the next iteration.

After performing the $z=0$ plane reflection painting tasks, the *DisplayObjectsInScene* checks the parameter *NumMirrors*, which indicates the number of reflecting surfaces. If it is one or less, you are through with the reflections. If it is greater than one, the reflection is defined to be in the $y=0$ plane. The proper sorting for that plane is performed, and the reflections are painted on the screen. Then the reflections are defined to be in the $x=0$ plane and the process is repeated. Next, regardless of mode or of the setting of *ReflSurface*, reflections are defined to be in no planes (actual objects being processed) and the proper sorting is performed, if indicated, for actual objects. The procedure *DisplayActualImages* is called to paint the actual images on the screen. This procedure is like *DisplayReflections* except that it processes the actual objects rather than reflections. When all objects have been displayed on the screen, the procedure *DisplayObjectsOnScreen* is complete.

The Model. Pas Description File Maker

When you run the *Model.Pas* program to generate solid modeled scenes, there are several options built into the program which are selected by the program reading the *Model.Des* file. These options are selected by running the *DesMaker.Pas* file before running the *Model.Pas* file. The contents of this program and the file that it makes are relatively simple. The program and its options are described in this chapter. The program itself and associated function are in the file *DesMaker.Pas*, listed in Figure 7-1.

First let's look at the function *Response*, which is designed to obtain a response to a question and make sure that it is either a *Y* or an *N*. It does this by means of two nested *Repeat* loops. The inner loop keeps trying to read a key from the keyboard until the response is other than a null, indicating that a key has been pressed. Then the character received from the keyboard is converted to a capital letter. If it is a *Y* or an *N*, the outer loop terminates; otherwise the loops continue to iterate until an acceptable response is received. Note that whenever *Response* is called in a program, nothing happens until you type in either *Y*, *y*, *N*, or *n*. When a correct response is received, if it is *Y*, the function returns *True*; otherwise, it returns *False*.

Now, let's look at the main program. First it displays the heading "Program to Generate a Description File for Model," followed by a blank line. Then it opens a file called *Model.Des* and prepares it for writing. It next asks, "Display Axis and Palette?," and writes the response to this query to a file line. You will see in the next chapter how this information decides if the entire palette and the coordinate axes are displayed on the screen along with the scene. Next, the program asks the question, "Display Solid Model?" The response is written to the next line of the file, indicating if solid facets are to be displayed to comprise the scene. If the response to this query

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

is *True* (solid facets are the display mode), the program writes *False* to the next two program lines (which indicates that the vertex and wire frame modes of display are not to be used). Otherwise, the program asks the question, "Display Preview of Vertices?" The response to this is written to the next line of the file if the vertex mode of display has been selected. The program asks the question, "Display Wire Frame Model?" The response to this query is written to the next line of the file if the wire frame mode of display has been selected.

Finally, regardless of display modes chosen, the program asks the question, "Load Scene from Disk?" The response to this query is written to the next line of the file. It will determine if a .SCN file will provide the display data or if the data will come from a procedure. After this, the disk file is closed, completing the operation of the program.

```
{
```

```
    Program for Generation of Description Files for Model.Pas
```

```
}
```

```
Uses
```

```
    Crt;
```

```
{
```

```
    Keyboard Response Routine
```

```
Response - returns true or false to a Y or N keyboard stroke  
}
```

```
Function Response : Boolean;
```

```
Var
```

```
    ch, c : Char;
```

```
Begin
```


THE MODEL.PAS DESCRIPTION FILE MAKER

```
Repeat
  Repeat
    ch := ReadKey
  Until (ch<>#0);
  c := UpCase(ch);
  Until (c='Y') Or (c='N');
  WriteLn(c);
  If (c='Y') Then
    Response := True
  Else
    Response := False
End;

{


Main Program


}

Type
  Name = String[32];

Var
  TextDiskFile : Text;
  BoolResp : Boolean;
  FileName : Name;

Begin
  ClrScr;
  FileName := 'Model.Des';
  WriteLn('Program to Generate a Description File for Model');
  WriteLn;
  Assign(TextDiskFile, FileName);
  Rewrite(TextDiskFile);

  Write('Display Axis and Palette? ');
  WriteLn(TextDiskFile, Response);

  Write('Display Solid Model? ');
  BoolResp := Response;
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
WriteLn(TextDiskFile,BoolResp);
If BoolResp Then
Begin
    WriteLn(TextDiskFile,Not(BoolResp));
    WriteLn(TextDiskFile,Not(BoolResp))
End
Else
Begin
    Write('Display Preview of Verticies? ');
    WriteLn(TextDiskFile,Response);
    Write('Display Wire Frame Model? ');
    WriteLn(TextDiskFile,Response)
End;

Write('Load Scene from Disk? ');
WriteLn(TextDiskFile,Response);

Close(TextDiskFile)
End.
```


The Solid Modeling Program

Now that you have looked at some of the elements needed for solid modeling, consider the solid modeling program itself. In this chapter assume that you already have a data file ending in the extension *.Scn* which provides the data needed to define a scene. In later chapters you'll see how to create such a data file. To get you started, there are thirteen data files for solid modeling included in Appendix A. The first of these is *CubePlan.Scn*, (Plate 6). The scene produced by this file is a cube on a plane. If you don't specify a file when asked, this is the file you will get by default. The next is *SphrPlan.Scn*. The scene produced by this file is a sphere on a plane. The third is *SphrWall.Scn*, (Plate 7). The scene produced by this file is a sphere in a mirrored room. The next file is *StakTors.Scn* (Plate 8). The scene produced by this file is a stack of three toroids. The next file is *FourCols.Scn*, (Plate 9). The scene produced by this file consists of four columns, each with a sphere on top. The next file is *Well.Scn*, (Plate 10) This scene was inspired by the "Old Well" at the University of North Carolina in Chapel Hill, NC. The next file is *Shapes.Scn*, (Plate 11). The scene produced by this file consists of a variety of objects to demonstrate the different shapes that can be produced. The next file is *Molecule.Scn* (Plate 12). The scene produced by this file is a picture of a water molecule. The next four files demonstrate the use of solid modeling to produce plots of equations. These are the files *PlotEqn1.Scn*, *PlotEqn2.Scn*, *PlotEqn3.Scn*, and *PlotEqn4.Scn*, illustrated in Plates 13, 14, 15, and 16, respectively. The last file is *SolOfRev.Scn* (Plate 17). The scene produced by this file contains a number of interesting shapes produced by solids of revolution. Another option covered later, allows you to generate a scene directly from a procedure rather than a stored file. If you select this option, the displayed

picture will be of stacked spheres, as shown in Plate 18. All of the .SCN data files are listed in Appendix A, so that you can copy them into your system if you didn't buy the program disk.

The program which takes one of these .SCN files and creates a scene on your display screen is *Model.Pas*. It and its associated procedures are listed in Figure 8-2.

Viewer and Light Source Vector

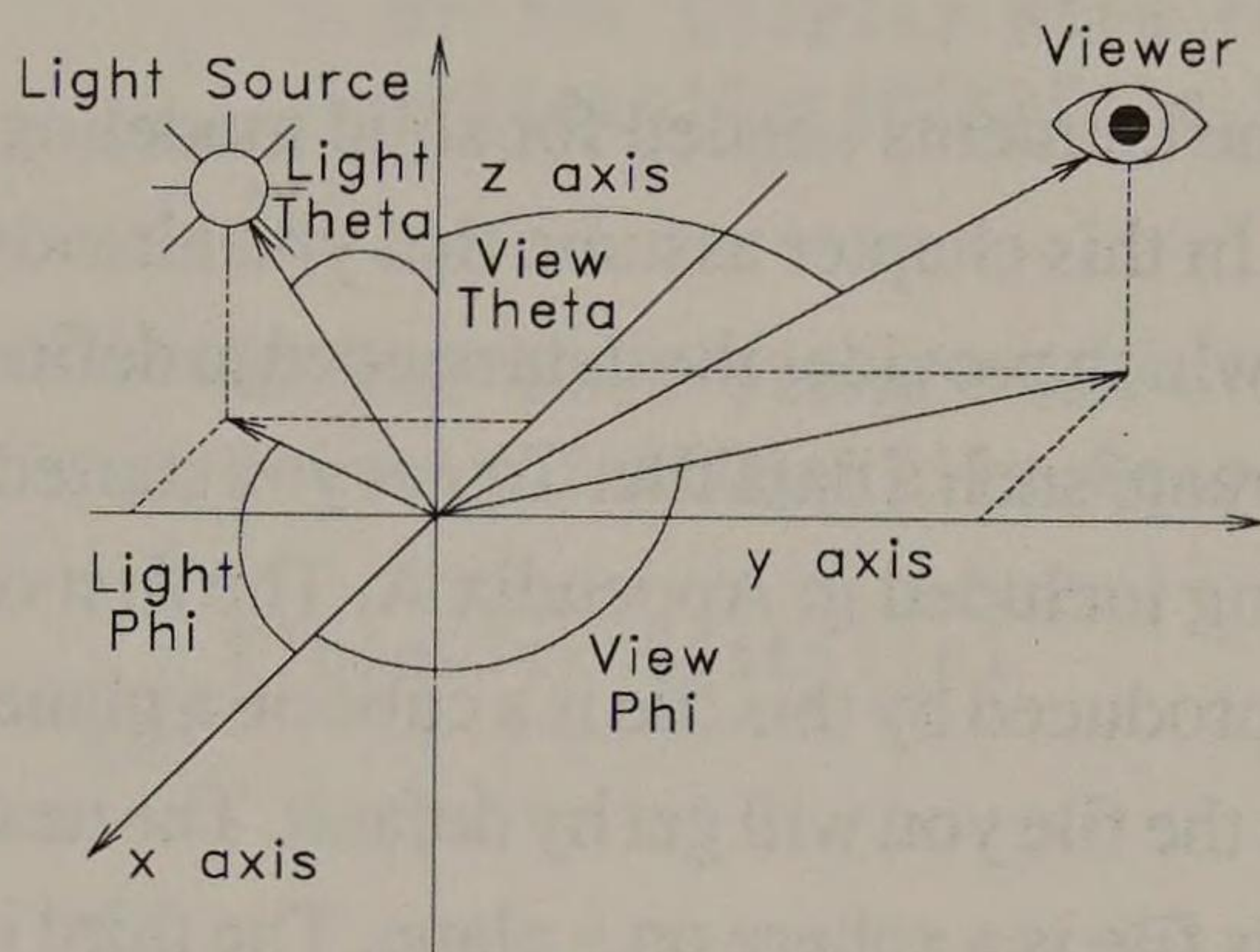


Figure 8-1. Viewer and Light Source Geometry

The position of the viewer determines what the scene is on the display screen. The viewer position is defined by two angles obtained from the file of scene information. These are *ViewPhi*, the angle between the viewer position and the *x-y* plane, and *ViewTheta* the angle between the viewer position and the *z* axis. The geometry is shown in Figure 8-1. These angles (in degrees) are passed to the procedure *GetViewVector*. This procedure first converts the angles to radians and determines the *x*, *y*, and *z* coordinates of a unit vector from the origin pointing in the viewer direction.

There are similar angles *LightPhi* and *LightTheta* which define the position of the light source illuminating the scene. This determines which part of an object will be brighter or dimmer because of the decrease in light intensity.

THE SOLID MODELING PROGRAM

{

Solid Modeling and Shading Routines for
Objects Constructed of Facets
By Christopher D. Watkins

}

Uses

Dos, Crt;

{ \$I Math.Inc }

{ \$I Graph.Inc }

{ \$I Model.Inc }

{

Viewer and Light Source Vectors

GetViewVector - unit vector in direction of viewer
InitLightDirection - get direction for light source
GetLightVector - unit vector in direction of light source

}

Const

ViewPhi = 270;

ViewTheta = 90;

Var

View : TDA;

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
Procedure GetViewVector;           { viewer unit vector 'v' }
```

```
  Var
```

```
    Phi, Theta : Real;
```

```
    x, y, z   : Real;
```

```
  Begin
```

```
    Phi := Radians( ViewPhi - Angl );
```

```
    Theta := Radians( ViewTheta - Tilt );
```

```
    x := Sin( Theta ) * Cos( Phi );
```

```
    y := Sin( Theta ) * Sin( Phi );
```

```
    z := Cos( Theta );
```

```
    Vec(x, y, z, View)
```

```
  End;
```

```
Var
```

```
  LightPhi : Real;
```

```
  LightTheta : Real;
```

```
Procedure InitLightDirection(LgtPhi, LgtTheta : Integer);
```

```
  Begin
```

```
    WriteLn('Light Direction is ', LgtPhi:4,
```

```
            '° around the z-Axis and');
```

```
    WriteLn('          ', LgtTheta:4,
```

```
            '° off of the z-Axis');
```

```
    LightPhi := LgtPhi;
```

```
    LightTheta := LgtTheta
```

```
  End;
```

```
Var
```

```
  Light : TDA;
```

```
Procedure GetLightVector;           { light unit vector 'l' }
```

```
  Var
```


THE SOLID MODELING PROGRAM

```
Phi, Theta : Real;  
x, y, z : Real;
```

```
Begin  
  Phi := Radians( LightPhi );  
  Theta := Radians( LightTheta );  
  x := Sin( Theta ) * Cos( Phi );  
  y := Sin( Theta ) * Sin( Phi );  
  z := Cos( Theta );  
  Vec(x, y, z, Light)  
End;
```

```
Var  
  Preview      : Boolean;  
  WireFrame    : Boolean;  
  SolidModel   : Boolean;  
  VerticalSort : Boolean;
```

```
{ $I AddObjs.Inc }
```

```
{
```

Load Description File and Scene Data

LoadDescAndScene - loads description file '.DES' and scene data

```
}
```

```
Procedure LoadDescAndScene(FileName : Name);
```

```
  Type  
    Strg = String[5];
```

```
  Var  
    B : Strg;
```

```
  Function Bool(B : Strg) : Boolean;
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
Begin
  If (B = 'FALSE') Then
    Bool := False
  Else
    Bool := True
End;

Var
  SceneFile : Name;

Procedure GetSceneFile;

  Var
    x, y : Byte;

  Begin
    WriteLn;
    Write('Enter File Name => ');
    x := WhereX;
    y := WhereY;
    ReadLn(SceneFile);
    If (SceneFile = '') Then
      Begin
        SceneFile := 'CubePlan';
        GotoXY(x,y);
        Write(SceneFile)
      End;
    WriteLn;
    WriteLn
  End;

Begin
  Assign(TextDiskFile, FileName);
  Reset(TextDiskFile);

  ReadLn(TextDiskFile, B);          { Axis and Palette Toggle }
  PutAxisAndPalette(Bool(B));
  If Bool(B) Then
    WriteLn('Axis and Palette Display On')
  Else
```


THE SOLID MODELING PROGRAM

```
WriteLn('Axis and Palette Display Off');

ReadLn(TextDiskFile, B);          { Display Solid Model? }
SolidModel := Bool(B);
If Bool(B) Then
    WriteLn('Solid Model Display On')
Else
    WriteLn('Solid Model Display Off');

If SolidModel Then { Vertex Display And Wire Frame Toggles }
    Begin
        ReadLn(TextDiskFile);
        ReadLn(TextDiskFile);
        Preview := False;
        WireFrame := False;
    End
Else
    Begin
        ReadLn(TextDiskFile, B);          { Display Verticies? }
        Preview := Bool(B);
        If Bool(B) Then
            WriteLn('Vertex Display On')
        Else
            WriteLn('Vertex Display Off');

        ReadLn(TextDiskFile, B);          { Display Wire Frame? }
        WireFrame := Bool(B);
        If Bool(B) Then
            WriteLn('Wire Frame Display On')
        Else
            WriteLn('Wire Frame Display Off')
    End;

ReadLn(TextDiskFile, B); {Read Scene from Disk or Calculate}
Close(TextDiskFile);
If Bool(B)
    Begin
        WriteLn('Loading Scene from Disk File');
        GetSceneFile;
        AddObjectsToSceneFromDiskFile(SceneFile)
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
End
Else
Begin
  WriteLn('Loading Scene from Procedure');
  WriteLn;
  AddObjectsToSceneFromProcedure
End;
Delay(2500)
End;
{
```

Affine Transformations

AffineTransformation - translate, rotate and scale verticies for
viewing

InvAffineTransformation - return verticies to original state
}

Procedure AffineTransformation(Tx, Ty, Tz, Sx, Sy, Sz, Rx, Ry, Rz
: Real);

Var

XForm : Matx4x4;

Temp : TDA;

Begin

PrepareMatrix(Tx, Ty, Tz, Sx, Sy, Sz, Rx, Ry, Rz, XForm);

For VertexNum := 1 To LastVertex Do

Begin { scale down from integer }

VecScalMultI(InvScaleData, Vertex[VertexNum], Temp);

Transform(Temp, XForm, Temp);

VecScalMultInt(ScaleImage, Temp, Vertex[VertexNum])

{ scale data }

End

End;

THE SOLID MODELING PROGRAM

```
Procedure InvAffineTransformation(Tx, Ty, Tz, Sx, Sy, Sz, Rx,
                                Ry, Rz : Real);
```

```
Var
```

```
  XForm : Matx4x4;
```

```
  Temp : TDA;
```

```
Begin
```

```
  PrepareInvMatrix(-Tx, -Ty, -Tz, 1/Sx, 1/Sy, 1/Sz, -Rx, -Ry,
                  -Rz, XForm);
```

```
  For VertexNum := 1 To LastVertex Do
```

```
    Begin { scale down from display }
```

```
      VecScalMultI(InvScaleImage, Vertex[VertexNum], Temp);
```

```
      Transform(Temp, XForm, Temp);
```

```
      VecScalMultInt(ScaleData, Temp, Vertex[VertexNum])
```

```
      { scale to integer }
```

```
    End
```

```
End;
```

```
{
```

Surface Normal Vector

```
  GetSurfaceNormalVector - unit vector normal to surface
```

```
}
```

```
Var
```

```
  SrfNorm : TDA;
```

```
Procedure GetSurfaceNormalVector(Face3d : VoxelArray);
```

```
Var
```

```
  Length : Real;
```

```
  Dir1 : TDA;
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
Dir2 : TDA;
Temp1 : TDA;
Temp2 : TDA;
Temp3 : TDA;
SrfNorm2 : TDA;
Length2 : Real;

Begin
  VecCopy(Face3d[2], Temp1);
  VecCopy(Face3d[1], Temp2);
  VecCopy(Face3d[LastVertexNumInFacet], Temp3);
  VecSub(Temp1, Temp2, Dir1);
  VecSub(Temp3, Temp2, Dir2);
  VecCross(Dir1, Dir2, SrfNorm);
  Length := VecLen(SrfNorm);

  VecCopy(Face3d[LastVertexNumInFacet], Temp1);
  VecCopy(Face3d[LastVertexNumInFacet-1], Temp2);
  VecCopy(Face3d[LastVertexNumInFacet-2], Temp3);
  VecSub(Temp1, Temp2, Dir1);
  VecSub(Temp3, Temp2, Dir2);
  VecCross(Dir1, Dir2, SrfNorm2);
  Length2 := VecLen(SrfNorm2)

  If (Length = 0) Then
    VecScalMult(1 / Length2, SrfNorm2, SrfNorm) { normalize
surface normal }
  Else
    If (Length2 = 0) Then
      VecScalMult(1 / Length, SrfNorm, SrfNorm)
    Else
      Begin
        VecScalMult(1 / Length, SrfNorm, SrfNorm);
        VecScalMult(1 / Length2, SrfNorm2, SrfNorm2);
        VecAdd(SrfNorm, SrfNorm2, SrfNorm);
        VecScalMult(0.5, SrfNorm, SrfNorm)
      End
    End;
End;
```


THE SOLID MODELING PROGRAM

The Illumination Model

Intensity - calculated intensity of point on screen

Function Intensity : Byte;

Const

Ambient = 0.30; {percentage of ambient light}

DifRfl = 0.50; {percentage of diffuse light from
reflection}

SpcRfl = 0.20; {percentage of specular light from
reflection}

Gloss = 5; {shininess }

Var

CosTheta : Real; { cosine of angle between 'n' and 'l' }

CosAlpha : Real; { cosine of angle between 'v' and 'r' }

TwoCosTheta : Real;

Ref : TDA;

Temp : TDA;

Begin

CosTheta := VecDot(SrfNorm, Light); { $\cos \theta = \mathbf{n} \cdot \mathbf{l}$ }

If (CosTheta < 0) Then

Intensity := Round(MaxInten * Ambient)

Else

Begin

TwoCosTheta := 2 * CosTheta;

VecScalMult(TwoCosTheta, SrfNorm, Temp);

VecNormalize(Temp);

VecSub(Temp, Light, Ref);

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
    VecNormalize(Ref)
    CosAlpha := VecDot(View, Ref); {  $\cos \alpha = v \cdot r$  }
    Intensity := Round( MaxInten * ( Ambient + DifRfl *
        CosTheta + SpcRfl * Power( CosAlpha , Gloss ) ) )
    End
End;
```

```
{
```

Facet Visibility Test

Visible - determine if facet is visible

```
}
```

Var

```
    Reflect : Boolean;
    MirrorX  : Boolean;
    MirrorY  : Boolean;
    MirrorZ  : Boolean;
```

Function Visible(Face3d : VoxelArray) : Boolean;

Var

```
    CosBeta      : Real;
    nvx, nvz, nvz : Real;
    Temp, V       : TDA;
```

Begin

```
    GetSurfaceNormalVector(Face3d);
```

```
    VecCopy(View, V);
```

```
    If Not(Preview Or WireFrame) And Reflect And MirrorZ Then
```

```
        V[2] := -V[2];
```

```
    If MirrorX Then
```

```
        V[0] := -V[0];
```


THE SOLID MODELING PROGRAM

```
If MirrorY Then
    V[1] := -V[1];
CosBeta := VecDot(SrfNorm, V);
If CosBeta < 0 Then
    Visible := False
Else
    Visible := True;
End;
```

```
{
```

Reflection Screen Buffer

Stores the reflective states of all screen pixel locations

InitReflectionBuffer - clear reflective states
Reflected - indicates whether or not reflective
MakeReflected - makes a screen location reflective

```
}
```

```
Const
    XBytes = 39; { (MaxX Div 8) - 1 bytes }
```

```
Var
    Refl : Array[0..XBytes, 0..MaxY] Of Byte;
```

```
Procedure InitReflectionBuffer;
```

```
Var
    I, J : Integer;
```

```
Begin
    For I := 0 To XBytes Do
        For J := 0 To MaxY Do
            Refl[I,J] := 0
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

End;

Function Reflected(x, y : Integer) : Boolean;

Begin

 If (Refl[x div 8, y] And (\$80 ShR (x Mod 8)) = 0) Then

 Reflected := False

 Else

 Reflected := True

End;

Procedure MakeReflected(x, y : Integer);

Begin

 Refl[x div 8, y] := Refl[x div 8, y] Or (\$80 ShR (x Mod 8))

End;

{

Polygonal Facet Fill routine

GetProjectedCoords - 3D point mapped onto 2D screen

PutFacet2 - fills a facet

}

Procedure GetProjectedCoords(Face3d : VoxelArray; Var Face2d : PixelArray);

Var

 xt, yt, zt : Real;

Begin

 For VertexNumInFacet := 1 To LastVertexNumInFacet Do

 Begin

 UnVec(Face3d[VertexNumInFacet], xt, yt, zt);

 If Reflect Then

 Begin

THE SOLID MODELING PROGRAM

```
    If MirrorZ Then
        Begin
            zt := -zt;
            If EdgeReflector And
                Not(EdgeReflectorAtZero)
                Then
                    zt := zt - 200 { 2 * -100 }
            End
        Else
            If MirrorY Then
                Begin
                    yt := -yt;
                    If EdgeReflector Or
                        EdgeReflectorAtZero
                        Then
                            yt := yt - 200 { 2 * -100 }
                    End
                Else
                    If MirrorX Then
                        Begin
                            xt := -xt;
                            If EdgeReflector Or
                                EdgeReflectorAtZero
                                Then
                                    xt := xt - 200 { 2 * -100 }
                            End
                        End
                    End;
                MapCoordinates(xt, yt, zt,
                    Face2d[VertexNumInFacet].x,
                    Face2d[VertexNumInFacet].y)
            End;
            Face2d[LastVertexNumInFacet+1] := Face2d[1]
        End;
    End;
```

Var

Mirroring : Boolean;

Procedure PutFacet2(Face2d : PixelArray; Color, Intens :

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

Byte);

Var

mnx, mxx, yc : Integer;

Procedure DrawHorizLine;

Var

xc : Integer;

Begin

For xc := mnx To mxx Do

Begin

If Mirroring Then

Begin

MakeReflected(xc, yc);

PutPixel(xc, yc, Color, Intens)

End

Else

If Not Reflect Then

PutPixel(xc, yc, Color, Intens)

Else

If Reflected(xc, yc)

PutPixel(xc, yc, Color, Intens)

End

End;

Var

i, x : Integer;

OldVertNum : Integer;

VertNum : Integer;

mny, mxy : Integer;

slope : Real;

Begin

If (Intens < 1) Then Exit;

If Reflect And Not(Intens = 1) Then { creates a darker
reflection }

Begin

Intens := Intens - 1;

If Not(Intens = 2) Then

THE SOLID MODELING PROGRAM

```
Begin
    Intens := Intens - 1;
    If Not(Intens = 3) Then
        Intens := Intens - 1
    End
End;

mny := Face2d[1].y;
mxy := Face2d[1].y;
For i := 2 To LastVertexNumInFacet Do { find min and max y
                                         coords }
    Begin
        If (Face2d[i].y > mxy) Then mxy := Face2d[i].y;
        If (Face2d[i].y < mny) Then mny := Face2d[i].y
    End;
    If (mny < 0) Then mny := 0;
    If (mxy > MaxY) Then mxy := MaxY;
    For yc := mny To mxy Do
        Begin
            mnx := MaxX + 1;
            mxx := -1;
            OldVertNum := LastVertexNumInFacet;
            For VertNum := 1 To LastVertexNumInFacet Do
                Begin
                    If (Face2d[OldVertNum].y >= yc) Or
                        (Face2d[VertNum].y >= yc)
                    Then
                        If (Face2d[OldVertNum].y <= yc) Or
                            (Face2d[VertNum].y <= yc)
                        Then
                            If Not(Face2d[OldVertNum].y =
                                Face2d[VertNum].y)
                            Then
                                Begin
                                    slope := (Face2d[VertNum].x -
                                        Face2d[OldVertNum].x) /
                                        (Face2d[VertNum].y -
                                        Face2d[OldVertNum].y);
                                    x := Round(slope * (yc -
                                        Face2d[OldVertNum].y))
                                        + Face2d[OldVertNum].x;
                                    If (x < mnx) Then mnx := x;
                                    If (x > mxx) Then mxx := x
                                End;
                                OldVertNum := VertNum
                            End
                        End
                    End
                End
            End
        End
    End
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
        End;
        If (mnx < 0) Then mnx := 0;
        If (mxx > MaxX) Then mxx := MaxX;
        If (mnx <= mxx) Then DrawHorizLine
    End
End;

{$I DispObjs.Inc}

{
    MAIN PROGRAM
}

Begin
    XRES:=320; YRES:=200;
    Title;
    WriteLn('Solid Modeling Program');
    WriteLn;
    InitReflectionBuffer; { Initialize Screen Reflection Buffer }
    InitVertexBuffer;     { Initialize Vertex Database }
    InitObjectBuffer;     { Initialize Object Database }
    LoadDescAndScene('Model.Des'); { Load Description and Scene
                                    Data }

    InitGraphics;

    AxisAndPalette; { Display Coordinate Axis and Palette }
    If DrawAxisAndPalette WaitForKey;

    DisplayObjectsInScene; { Display the Objects }

    ExitGraphics
End.
```

Figure 8-2. Listing of the *Model.Pas* File

These angles (in degrees) are passed as parameters to the procedure *InitLightDirection*, which first displays a description of the two angles, their values, and loads the values into the two previously described parameters. Another procedure, *GetLightVector*, converts these angles to radians and determines the x , y , and z coordinates of a unit vector starting at the origin and pointing in the direction of the light source.

Loading the Description File and Scene Data

The procedure *LoadDescAndScene* loads all of the information that generates the scene with the desired options. The parameter passed to this procedure is the name of the description file, which is usually *Model.Des*. The procedure opens this file and reads the first line, which indicates if the coordinate axes and the palette should be displayed upon the screen. The file line contains either the string "TRUE" or the string "FALSE." The function *Bool* returns *False* if the string is "FALSE" and *True* otherwise. This Boolean value passes to the function *PutAxisAndPalette*, which sets the parameter for this option. If the Boolean value was *True*, the procedure next writes the legend, "Axis and Palette Display On" to the screen. Otherwise, it writes the line, "Axis and Palette Display Off." The next line determines if the display is to show solid models. This is determined by the parameter *SolidModel*, which is set to the Boolean value obtained from disk file line and passing it through *Bool*. If the value is *True* the line "Solid Model Display On" is written to the display. Otherwise, "Solid Model Display Off" is written. If the value is *True*, the next two lines are read from the file, but disregarded, and the option parameters *Preview* and *WireFrame* are set to *False*. If the value is *False*, the procedure reads the next line of the disk file and sets *Preview* to the Boolean value specified by the file line. It displays the line "Vertex Display On," or "Vertex Display OFF," as appropriate. The procedure reads the next line of the disk file and sets *WireFrame* to the Boolean value specified by the file line. It displays the line "Wire Frame Display On," or "Wire Frame Display OFF," as appropriate.

Once *SolidModel*, *Preview*, and *WireFrame* are set to the proper values, and the corresponding display output to the screen, the procedure reads the next line from the disk file, indicating if the scene data is from a disk file or a procedure. The procedure

then closes the disk file. If the selected option is to read data from a procedure, the line "Loading Scene from Procedure" is displayed and *AddObjectsFromProcedure*, which was described in Chapter 5, is called to load the scene data. If the selected option is to read data from a disk file, the procedure *GetSceneFile* is called. This procedure displays, "Enter File Name =>" and saves the cursor position. Next, the keyboard is read to the parameter *SceneFile*. If only a carriage return is entered, the file name *CubePlan* is placed in the parameter *SceneFile* by default, the cursor repositions at the end of the displayed string, and the default file name is displayed on the screen. This is the end of the *GetSceneFile* procedure so return to the *LoadDescAndScene* procedure, which calls *AddObjectsToSceneFromDiskFile*, described in Chapter 6, to load the scene data from the file named in *SceneFile*. After the scene data loads, from whatever source, a 2.5 second delay assures that the user has time to read all pertinent information from the screen. The *LoadDescAndScene* procedure ends.

Affine Transformations

Given that you have defined a facet by specifying its vertices, you'll now need to transform it from a generic position, size, and orientation in a generic object file, to the position, size, and orientation you'll want in the particular scene you are creating. The procedure *AffineTransformation* performs these operations. It begins by calling *PrepareMatrix* (in Chapter 1) to create a transformation matrix from the current translation, scaling, and rotation coordinates. It executes a *For* loop for every vertex comprising the facet. Each iteration of the loop begins by scalar multiplication of a vertex vector, to convert it to the +23000 to -23000 integer system that speeds up calculations. The procedure *Transform* (Chapter 1) multiplies the vertex by the transformation matrix. Next an integer vector, scalar multiplication converts to the screen-display coordinate system.

The procedure *InvAffineTransformation* is the opposite of *AffineTransformation*. It takes a set of vertices representing the final position of a facet in display coordinates (the result obtained by running *AffineTransformation*) and runs all of the operations in reverse order to obtain the original set of coordinates for each facet vertex.

Finding the Surface Normal Vector

One critical factor in finding the shade to be painted on the screen for a particular facet is the surface normal vector to the facet. The procedure *GetSurfaceNormalVector* generates this vector. It begins by finding two vectors, the first drawn from the first vertex of the facet to the second vertex, and the second drawn from the last vertex to the second vertex. It takes the vector cross-product of these two vectors. If non-zero, this is in the direction of the surface normal. The length of this vector is found. The procedure then repeats the same set of operations using the last three vertices of the facet and again finding a vector cross-product and its length. If the length of the first surface normal vector is zero, the second surface normal vector is normalized and returned by the procedure. If the length of the second surface normal vector is zero, the first surface normal vector is normalized and returned by the procedure. Otherwise both surface normal vectors are normalized, the normalized vectors are added together and the result divided by two and returned by the procedure.

The Illumination Model

The illumination model determines the shade to paint a particular facet to the screen. This is performed by the function *Intensity*, which returns a byte that represents which of the 256 available colors paints the facet to the screen. The function defines the percentage of illumination supplied by ambient, diffuse, and specular light, as well as the power for shininess in a Phong shading model. This model says that the shininess of a facet is defined by:

$$I = s \cos^g \alpha$$

(Equation 8-1)

where I is the intensity of the light at the surface, s is the spectral reflection coefficient of the facet, α is the angle between the viewer vector and facet, and g is the shininess factor. The procedure finds the cosine of the angle between the surface normal vector and the light source vector (theta). If this cosine is negative, there is no reflected light, so the ambient light value is returned. If the cosine is positive, the function multiplies the surface normal vector by twice the cosine found above, normalizes the final resulting vector, subtracts the light vector from it, and normalizes the resulting vector. The dot product of this vector and the *View* (viewer direction) vector is taken

to find the cosine of the angle between the two vectors (alpha). The value returned by the function in this case is the maximum intensity value (255) multiplied by the ambient light fraction plus the diffuse light fraction, multiplied by the cosine of theta plus the specular reflection, multiplied by the cosine of alpha raised to the *Gloss* (shininess) power.

Testing a Facet for Visibility

The procedure *Visible* determines whether a facet is visible to the viewer or not. Although this procedure is listed as part of the *Model.Pas* file, it is actually used by the procedure *PlaceObjectOnScreen*, described in Chapter 6. This procedure is passed the array containing the coordinates of the vertices of a facet. It begins by calling *GetSurfaceNormal* to obtain the surface normal vector for the facet. Next, the view vector is copied to a new location. Take the dot product of these two vectors to determine the angle between them. First, however, you have to reverse the direction of one of the coordinates of a vector to make the direction correct if you are dealing with a reflection instead of the object. The procedure therefore uses a series of *If* statements to check which mirror plane is set and adjust the vector coordinates, accordingly. It takes the dot product of the two vectors to form the cosine of beta. If this cosine is negative, the facet is invisible, so the function returns *False*. Otherwise, the facet is visible and the function returns *True*.

The Reflection Screen Buffer

In order to keep track of the reflective state of every pixel on the screen, you'll need a buffer which contains one bit for every screen pixel. This buffer is *Refl*. It consists of an array of *x-resolution-divided-by-eight-bytes-by-y-resolution* rows. Two procedures and a function are associated with this buffer. The procedure *InitReflectionBuffer* uses a pair of nested *For* loops to set every member of the array to zero. The function *Reflected* isolates the bit representing a particular pixel in the array. If it is zero, the function returns *False*; otherwise it returns *True*. The procedure *MakeReflected* sets a designated bit (in the proper byte in the array) to one for *True*.

Obtaining Facet Screen Coordinates

The procedure *GetProjectedCoords* is passed an array *Facet3d* which comprises the three-dimensional coordinates of each of the vertices that form the facet definition. For each vertex, the procedure decomposes the vertex vector into three separate coordinates. It then looks at the mirror parameters to see if one of the reflections of the object is being processed. If this is the case, the coordinates are modified to represent the coordinates of the reflection rather than the object. An *If* statement is used to determine the positioning of the edge reflector, which is reflecting the object, and the proper coordinate conversion is performed for whichever edge reflector is being used. Once the coordinate conversions (if any are required) are complete, the procedure *MapCoordinates* converts from three-dimensional coordinates to two-dimensional coordinates representing a position on the display screen surface. These are placed in a two-dimensional vector *Facet2d[n]*, where *n* is the number of the vector being processed. When all vertices have been processed, the procedure is complete. The vertex coordinates for the two-dimensional representation of the facet on the screen are in the array *Facet2d*.

Painting a Solid Facet on the Screen

The procedure *PutFacet2* fills in a facet on the screen with the designated color and intensity. It is very similar to *PutFacet* (Chapter 4), except it takes mirroring into account. It begins by checking if the parameter *Intens* is large enough for the facet to be visible, and if not, the procedure terminates. Next, the procedure checks if an object or reflection is being processed. If a reflection is being processed, the intensity decreases to make it darker than the actual object. Next, the procedure loops to determine minimum and maximum values of the *y* coordinate for the facet. A *For* loop iterates once for every value of *y* within the facet. Within the loop, the procedure computes the slope of the lines connecting critical pairs of vertices and uses the slopes to determine beginning and ending values of *x* for the current value of *y*. These values are tested to assure they are never smaller than the minimum *x*, nor larger than the maximum *x* for the facet. If the maximum is greater than or equal to the minimum, the procedure *DrawHorizLine* draws a line of the proper color for that *y* value. This procedure is different from the *DrawHorizLine* procedure used in *PutFacet* because

it takes mirroring into account. The procedure iterates for all values of x from the beginning to the end value at the desired value of y . If in the mirroring mode, it makes each pixel in the reflection data matrix *True* and uses *PutPixel* to paint the pixel on the screen with the proper color. If not in the mirroring mode, the procedure checks to see if you are processing a reflection. If not, it paints the pixel on the screen using *PutPixel*. If you are processing a reflection, it checks to see if this particular pixel location is capable of showing a reflection; if it is, the procedure paints the pixel on the screen using *PutPixel*. After all x values are processed, the *DrawHorizLine* procedure is complete. *PutFacet2* loops through another value of y until all values of y within the facet have their associated lines drawn, resulting in the facet being filled with the desired shade of color.

The *Model.Pas* Program

Now you are finally done with examining the necessary procedures and can proceed with describing the *Model.Pas* program. It begins by writing the legend "Solid Modeling Program" to the screen and initializes the reflection, vertex, and object buffers. Next, the procedure *LoadDescAndScene* is called. This procedure reads *Model.Des* to determine which options have been selected. It also sets up for the proper option of solid, vertex, or wire frame modeling. If adding a scene from a procedure was selected, the procedure loads the scene data. Otherwise, the name of the scene file is requested, and when given, the scene data loads from that file. Next, the program calls *InitGraphics* to initialize for graphics display. It calls *AxisAndPalette*, which displays the coordinate axes and the color palette on the screen. If that option was selected, the program waits for a keystroke before continuing. Finally, *DisplayObjectsInScene* displays all objects on the screen. The program now waits for another keystroke, and when one occurs, it exits the graphics mode and terminates.

Creating Object Databases

At this point you should know how to model a scene containing solid objects, if you have a file of solid object data or a procedure containing the data you need. Whichever technique defines scene data, the program makes use of a list of objects, each already defined in a file called at the appropriate point to display the object on the screen. In this chapter, you'll see how these object files are created, so you will understand what's in the file for each primitive object in this book. You'll also know how to create primitive objects of your own.

The file format for an object data file is described in Chapter 4, but let's go over it quickly here. The first line of the file consists of the number of vertices in the object, the number of facets in the object, and the number of vertices in each facet. The file is a text file, and these numbers are written in ASCII representation, separated by spaces. The next section of the file consists of a set of three integers, representing the x , y , and z coordinates, for each vertex. Each integer is written in binary form and constitutes two bytes. The next section of the file consists of a list of the numbers of the vertices that make up each facet, in order. Each is also a binary integer taking up two bytes. Keeping this firmly in mind, let's look at some procedures needed in creating data files. These procedures are included in the file *ShpMk.Inc.*, listed in Figure 9-1.

Adding Vertices

Perhaps you remember that the vertex information stored in a data file is in the form of an integer with values between -32767 and $+32,767$. This is a bit misleading because when you are creating the data file, the vertex information is in the form of real numbers between -1 and $+1$. The transformation to the other scale occurs within the software. The procedure that adds a vertex to the vertex list, and ultimately

to the object data file, is called *AddVertex*. This procedure passes three numbers, corresponding to the x , y , and z coordinates of the new vertex. It first checks that each of these is between -1 and $+1$. If any one is out of limits, an error parameter is set to *True*; otherwise each error parameter is set to *False*. If any coordinate was out of range, a sound is produced, followed by a message on the display telling which coordinate it was. The program halts, which causes you to return to DOS or Turbo Pascal, wherever you were when you ran the program. Next, the three coordinates are multiplied by the proper scale factor and converted to integers. The procedure looks at the parameter *RepeatedVertexCheck*. If this parameter is *True*, the procedure *CheckRepeatingVertices* is called. This procedure first sets the parameter *RepeatedVertex* to *False*. If this is not the first facet, the procedure checks each of the present vertex coordinates against the corresponding coordinate of other vertices in the list. If a set of coordinates matches, *RepeatedVertex* is set to *True* and the procedure terminates. If no match is found, the procedure ends with *RepeatedVertex* set to *False*. At the *AddVertex* procedure, if a matching vertex is reported, its number is stored as the present vertex number in the facet description, and the procedure ends. If a match was not reported, either because you weren't supposed to check for matching or because no match was found, the current vertex is stored in the vertex data array and scaled and plotted to the screen. Its number is stored as the present vertex number in the facet description, its number is set as the last vertex number, and the index for the next vertex is incremented.

Initializing Before Making Vertices

Before running any of the programs that create facets and vertices, some initialization needs to be done. The procedure *InitVertexMaker* performs these tasks. First the procedures *InitPerspective*, *InitPlotting*, and *InitGraphics* set up in the graphics mode for plotting a perspective scene. Next, *PutAxisAndPalette* sets up for displaying the axes and the color palette on the screen and *AxisAndPalette* displays this information. The procedure *InitVertexBuffer* clears the vertex buffer so it is ready to receive new data. Finally, *RepeatedVertexCheck* is set to *True* so that vertices will not be repeated in the list.

CREATING OBJECT DATABASES

{

Addition of Vertices to Objects

AddVertex - adds vertex to database

}

Var

RepeatedVertexCheck : Boolean;

RepeatedVertex : Boolean;

OldVertexNum : Integer;

Procedure CheckForRepeatingVerticies(x, y, z : Integer);

Begin

RepeatedVertexCheck := False;

If (FacetNum > 1) Then

For OldVertexNum := VertexNum-1 DownTo 1 Do

If (Vertex[OldVertexNum][0] = x) And

(Vertex[OldVertexNum][1] = y) And

(Vertex[OldVertexNum][2] = z) Then

Begin

RepeatedVertex := True;

Exit

End

End;

Procedure AddVertex(xr, yr, zr : Real);

Var

XError, YError, ZError : Boolean;

x, y, z : Integer;

sc : Real;

Begin

If (xr < -1) Or (xr > 1) Then

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
      XError := True
    Else
      XError := False;
    If (yr < -1) Or (yr > 1) Then
      YError := True
    Else
      YError := False;
    If (zr < -1) Or (zr > 1) Then
      ZError := True
    Else
      ZError := False;

    If XError Or YError Or ZError Then
      Begin
        Sound(1000);
        Delay(1000);
        NoSound;
        ExitGraphics;
        Write('Out of Range -1..1 in ');
        If XError Then
          WriteLn('X !')
        Else
          If YError Then
            WriteLn('Y !')
          Else
            WriteLn('Z !');
        Delay(2000);
        Halt
      End;

    x := Round(ScaleData * xr);
    y := Round(ScaleData * yr);
    z := Round(ScaleData * zr);
    If RepeatedVertexCheck Then
      CheckForRepeatingVertices(x, y, z)
    Else
      RepeatedVertex := False;
    If Not RepeatedVertex Then
      Begin
        VecInt(x, y, z, Vertex[VertexNum]);
        sc := 100 / ScaleData; { makes preview plot -200 to
                                200 on axis }
```


CREATING OBJECT DATABASES

```
CartesianPlot3D(sc * x , sc * y, sc * z, 215);
Facet[FacetNum,VertexNumInFacet] := VertexNum;
LastVertex := VertexNum;
VertexNum := VertexNum + 1
End
Else
Facet[FacetNum,VertexNumInFacet] := OldVertexNum;
VertexNumInFacet := VertexNumInFacet + 1
End;
```

```
{
```

Initialization of Vertex Database Maker

InitVertexMaker - calls routines required to generate an object database

```
}
```

Procedure InitVertexMaker;

Begin

InitPerspective(False, 0, 0, 500, 500);

InitPlotting(215,18);

InitGraphics;

PutAxisAndPalette(True);

AxisAndPalette;

InitVertexBuffer;

ReduceRepeatedVerticies(True) { calls database minimizer }

End;

Figure 9-1. Listing of ShpMk.Inc File

Creating Objects with the *MakeObj.Pas* Program

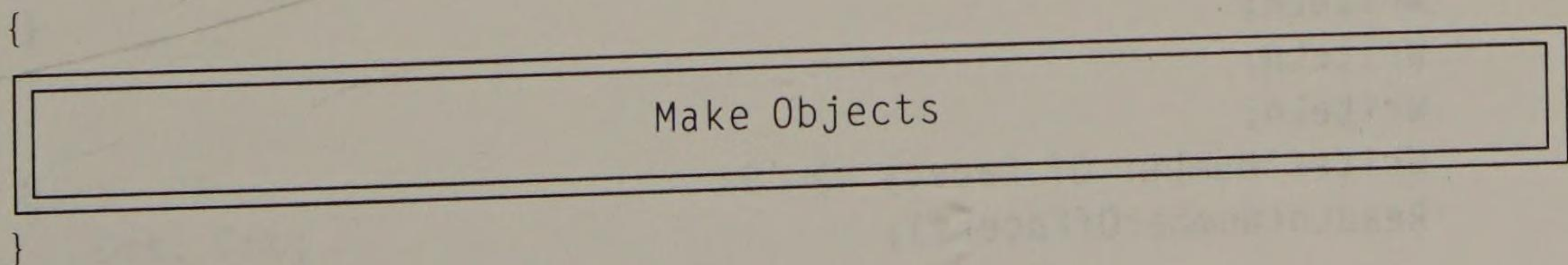
The *MakeObj.Pas* program gives you the opportunity to create an object data file with computer assistance. This program first asks you to name the data file, and asks for the number of facets. It displays the palette and axes on the screen and allows you to enter all of the necessary vertices (four for each facet). It plots each of these vertices on the screen and creates an object data file from them. The program is listed in Figure 9-2. It begins by displaying, "Make Object Databases", followed by a couple of blank lines, followed by, "Enter File Name => ". The program stores the current cursor position. At this point, type in the name you want to assign to the new object data file. It may be any acceptable eight-character DOS file name. It should not include an extension. If, at this point, only the *Ent* key is hit, the program does the following. First, it automatically supplies the file name *NewObj*. Then it returns to the stored cursor position and writes "NewObj" to the screen. If a file name is entered, it will already be displayed on the screen. The program adds to the file name the extension *.Dat*. Next, three lines are skipped and the program displays, "Number of Facets=>". Enter the number of facets that comprise the object. The procedure *InitVertexMaker*, described above, prepares to generate the facet information, including entering the graphics mode and displaying the coordinate axes and color palette on the screen. The program sets the values of some parameters. *VertexNum* and *VertexNumInFacet* are set to one. You are currently ready to enter data for the first vertex, which is also the first vertex in the facet. The program also initializes *LastFacet*, the number of the last facet with the number of facets that was entered above. The parameter *LastVertexNumInFacet* is set to four, so all facets in this program must have four vertices. (If you want a triangular facet, you can get around this very easily by specifying two sequential vertices to be at the same coordinates.) Finally, *LastVertex* is set to the product of the number of facets and the number of vertices per facet.

The program is now ready to process vertex data. For each of the facets that comprise your object, you need to enter a set of three coordinates for each of the four facets. Remember that each coordinate must be between -1 and +1. The coordinates are separated by a space, and after each set, the *Ent* key is hit. When you have entered all of the vertices, the program gives a beep. If you hit *Ent* again, the program calls

the procedure *SaveData*, described in Chapter 4, and saves your object to the designated file. The program calls *ExitGraphics* to return to the text display mode and ends.

This sounds simple, but it's more complicated than it seems. Although each vertex that you enter is displayed on the screen, unless you have quite a few of them, the display may not tell you very much. Furthermore, you need to have the coordinates of every vertex well-determined before beginning, and know precisely which vertices make up each facet and in what order they are entered. The problems with not having fully pinned down this data are as follows. First, if any one of your vertex coordinates is out of range (less than -1 or greater than $+1$) the program terminates and you have to fix the problem and begin again. Second, if you make a mistake on a line, you can correct it, but once you hit the *Ent* key, it's too late for that line. To correct the error, you have to quit and start again. Third, if you have an error in the vertex coordinates causing the vertex to appear in the wrong place on the screen, the same situation occurs, namely it's too late to correct so you need to quit and begin again. Finally, if all of your vertex information is correct, but you don't assign the correct vertices to a facet, or if you have the right vertices but not in the right order, when you finally use the data file in the *Model.pas* program to generate a scene, the solid is not the way you want it. You'll need to find your error and regenerate the file from scratch.

You can see from this that while manually entering the coordinates for each vertex of each facet is a feasible way to generate a data file for an object, some computer assistance in generating such files is highly desirable. In the next few sections, are described some computer programs for generating those data files. You'll need to use these files to generate the .dat files for the primitive objects that make up a scene before you are able to run the *Model.pas* program.



ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

Uses

Dos, Crt;

{\$I Math.Inc}

{\$I Graph.Inc}

{\$I Model.Inc}

{\$I ShpMk.Inc}

Var

ObjF : Name;

xs, ys : Byte;

NumberOfFacets : Integer;

T : Integer;

x, y, z : Real;

Begin

Title;

WriteLn('Make Object Databases');

WriteLn;

WriteLn;

Write('Enter File Name => ');

xs := WhereX;

ys := WhereY;

ReadLn(ObjF);

If (ObjF = '') Then

Begin

ObjF := 'NewObj';

GotoXY(xs,ys);

Write(ObjF)

End;

ObjF := ObjF + '.Dat';

WriteLn;

WriteLn;

WriteLn;

Write('Number Of Facets => ');

ReadLn(NumberOfFacets);


```

InitVertexMaker;
VertexNum      := 1;
VertexNumInFacet := 1;
LastFacet      := NumberOfFacets;
LastVertexNumInFacet := 4;
LastVertex     := LastFacet * LastVertexNumInFacet;
For FacetNum := 1 To LastFacet Do
Begin
  For T := 1 To LastVertexNumInFacet Do { create facet from
                                         vertices }

  Begin
    ReadLn(x, y, z);
    AddVertex(x, y, z)
  End;
  VertexNumInFacet := 1
End;
SaveData(ObjF);

ExitGraphics
End.

```

Figure 9-2. Listing of *MakeObj.Pas* Program

Generating a Cone or Pyramid Data File

The program *ConePyrm.Pas* generates the object data files for both a pyramid and a cone. This program is listed in Figure 9-3. It begins by calling *InitVertexMaker* to initialize before making an object data file. It calls the procedure *SetupCone*. The parameter passed to this procedure is the number of facets to be used.

```

{
  Cone and Pyramid Database Generator
}

```

```

Uses
  Dos, Crt;

```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
{ $I Math.Inc }
```

```
{ $I Graph.Inc }
```

```
{ $I Model.Inc }
```

```
{ $I ShpMk.Inc }
```

```
Var
```

```
  Theta, DTheta, HalfDTheta : Integer;
```

```
Procedure SetupCone(NumOfFacets : Integer);
```

```
  Begin
```

```
    VertexNum := 1;
```

```
    VertexNumInFacet := 1;
```

```
    LastFacet := NumOfFacets;
```

```
    LastVertexNumInFacet := 4;
```

```
    LastVertex := LastFacet * LastVertexNumInFacet;
```

```
    DTheta := 360 Div LastFacet;
```

```
    HalfDTheta := DTheta Div 2;
```

```
    Theta := HalfDTheta
```

```
  End;
```

```
Var
```

```
  NumFacetsOnEndCap, T : Integer;
```

```
Procedure EndCap;
```

```
  Begin
```

```
    NumFacetsOnEndCap := LastFacet Div 2 - 1;
```

```
    If NumFacetsOnEndCap And 1 = 1 Then
```

```
      Theta := -90 + HalfDTheta
```

```
    Else
```

```
      Theta := -90 + DTheta;
```

```
      If LastFacet = 4 Then
```


CREATING OBJECT DATABASES

```
NumFacetsOnEndCap := 2;
For T := 1 To NumFacetsOnEndCap Do { bottom cap }
Begin
  FacetNum := FacetNum + 1;
  LastFacet := LastFacet + 1;
  AddVertex( CosD(Theta+HalfDTheta),
              SinD(Theta+HalfDTheta), -1);
  AddVertex( CosD(Theta-HalfDTheta), SinD(Theta-
              HalfDTheta), -1);
  AddVertex( CosD(180-Theta+HalfDTheta), SinD(180-
              Theta+HalfDTheta), -1);
  AddVertex( CosD(180-Theta-HalfDTheta), SinD(180-Theta-
              HalfDTheta), -1);

  VertexNumInFacet := 1;
  Theta := Theta + DTheta
End
End;

Procedure MakeConeDatabase;

Begin
  For FacetNum := 1 To LastFacet Do
    Begin
      AddVertex( CosD(Theta-HalfDTheta), SinD(Theta-
          HalfDTheta), -1 );
      AddVertex( CosD(Theta+HalfDTheta),
          SinD(Theta+HalfDTheta), -1 );

      AddVertex( 0, 0, 1 );
      AddVertex( 0, 0, 1 );
      Theta := Theta + DTheta;
      VertexNumInFacet := 1
    End;
  EndCap
End;

{
  MAIN PROGRAM
}
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
Begin
  InitVertexMaker;

  SetupCone(4); { Num Facets => 360 Div Even Number = Integer }
  MakeConeDatabase;
  SaveData('Pyramid.Dat');

  SetupCone(180); { Num Facets => 360 Div Even Number = Integer
                  (180 is max) }

  MakeConeDatabase;
  SaveData('Cone.Dat');

  ExitGraphics
End.
```

Figure 9-3. Listing of Program to Generate Cone or Pyramid

This parameter must be an even number that divides into 360 without a remainder. At this first pass, generate the file *Pyramid.Dat*, to produce a pyramid. So, there are only four facets. The procedure *SetUpCone* first sets starting values into the variables *VertexNum*, *VertexNumInFacet*, *LastFacet*, *LastVertexNumInFacet*, and *LastVertex* in the program described above. It creates the parameters *DTheta* and *HalfDTheta* which represent angular increments around the base of the cone. The parameter *Theta* is set to the starting angular position. Getting back to the main program, the procedure *MakeConeDataBase* is called next. This procedure begins with a *For* loop which iterates for each facet. It adds the four vertices for the facet to the vertex list. These vertices are located at $(\cos(\Theta - \text{HalfD}\Theta), \sin(\Theta - \text{HalfD}\Theta), -1)$, $(\cos(\Theta + \text{HalfD}\Theta), \sin(\Theta + \text{HalfD}\Theta), -1)$, $(0, 0, -1)$, and $(0, 0, -1)$. Thus each facet is a triangle having a base that is a straight line segment approximating a short length of the circle (the base of the cone), and having an apex that is the apex of the cone. The procedure increments *Theta* by *DTheta*, resets *VertexNumInFacet* to one, and loops again until all facets have been processed. After the loop is complete, the procedure *EndCap* generates the cap at the base of the pyramid or cone. This procedure determines the number of facets on the end cap, which is one less than half the number of facets used for the rest of the cone surface.

It sets up the initial position of the angle *Theta*. Then a *For* loop runs which iterates once for each facet. Each facet is a trapezoid that has as its two sides straight line segments that approximate a short segment of the circle. When these trapezoids are stacked, they produce an approximation of the circular surface. At the end of each iteration of the loop, *VertexNumInFacet* is reset to one and *Theta* increases by *DTheta*. The procedure ends when the loop has completed its iterations, returning to *MakeConeDataBase*, which also ends, returning to the main program. The main program calls *SaveData* to create the pyramid object data file and save the vertex and facet data to it. The procedures *SetupCone*, *MakeConeDatabase*, and *SaveData* are recalled to repeat the entire process with 180 facets. The program returns to the text display mode and ends.

Generating the Data File for a Cylinder

The program *Cylinder.Pas* generates the object data file for a cylinder, and is listed in Figure 9-4. It begins by calling *InitVertexMaker* to do the initialization necessary before making an object data file. It calls the procedure *SetupCylinder*. The parameter passed to this procedure is the number of facets. The parameter must be equal to an even number that divides into 360 without a remainder. This program currently uses 90 facets. The procedure *SetUpCylinder* first sets starting values into the variables *VertexNum*, *VertexNumInFacet*, *Last Facet*, *LastVertexNumInFacet*, and *LastVertex* as in the program described above. It creates the parameters *DTheta* and *HalfDTheta*, which represent angular increments around the base of the cone. The parameter *Theta* is set to the starting angular position. Getting back to the main program, the procedure *MakeCylinderDataBase* is called next. This procedure begins with a *For* loop which iterates for each facet. It adds the four vertices for the facet to the vertex list. These vertices are located at $(\cos(\Theta - \text{HalfD}\Theta), \sin(\Theta - \text{HalfD}\Theta), -1)$, $(\cos(\Theta + \text{HalfD}\Theta), \sin(\Theta + \text{HalfD}\Theta), +1)$. Thus each facet is a rectangle having a base that is a straight line segment approximating a short length of the circle that is the base of the cylinder, and a top that is a straight line segment approximating a short length of the circle, that is the top of the cylinder. The procedure increments *Theta* by *DTheta*, resets *VertexNumInFacet* to one, and loops again until all facets have been processed. After the loop is complete, the

procedure *EndCaps* generates the caps at the top and bottom of the cylinder. This procedure first determines the number of facets that will be on the end cap, which is one less than half the number of facets used for the rest of the cone surface. It sets up the initial position of the angle Theta. Then a *For* loop is run, and iterates once for each facet. Each facet is a trapezoid that has as its two sides, straight line segments that approximate a short segment of the circle. When these trapezoids are stacked, they produce an approximation of the circular surface. At the end of each iteration of the loop, *VertexNumInFacet* is reset to one and Theta is increased by *DTheta*. The procedure ends when the loop completes its iterations, returning to *MakeCylinderDataBase*, which also ends, returning to the main program. The main program calls *SaveData* to create the cylinder object data file and save all of the vertex and facet data. The program returns to the text display mode and ends.

```
{
  

Cylinder DatabaseGenerator


}
```

Uses

Dos, Crt;

{ \$I Math.Inc }

{ \$I Graph.Inc }

{ \$I Model.Inc }

{ \$I ShpMk.Inc }

Var

Theta, DTheta, HalfDTheta : Integer;

Procedure SetupCylinder(NumOfFacets : Integer);

CREATING OBJECT DATABASES

```
Begin
  VertexNum      := 1;
  VertexNumInFacet := 1;
  LastFacet      := NumOfFacets;
  LastVertexNumInFacet := 4;
  LastVertex     := LastFacet * LastVertexNumInFacet;
  DTheta        := 360 Div LastFacet;
  HalfDTheta     := DTheta Div 2;
  Theta         := HalfDTheta
End;
```

```
Var
  NumFacetsOnEndCap, T : Integer;
```

```
Procedure EndCaps;
```

```
Begin
  NumFacetsOnEndCap := LastFacet Div 2 - 1;
  If NumFacetsOnEndCap And 1 = 1 Then
    Theta := -90 + HalfDTheta
  Else
    Theta := -90 + DTheta;
  For T := 1 To NumFacetsOnEndCap Do
    Begin
      FacetNum := FacetNum + 1; { top cap }
      LastFacet := LastFacet + 1;
      AddVertex( CosD(Theta-HalfDTheta),
                  SinD(Theta-HalfDTheta), 1);
      AddVertex( CosD(Theta+HalfDTheta),
                  SinD(Theta+HalfDTheta), 1);
      AddVertex( CosD(180-Theta-HalfDTheta),
                  SinD(180-Theta-HalfDTheta), 1);
      AddVertex( CosD(180-Theta+HalfDTheta),
                  SinD(180-Theta+HalfDTheta), 1);

      VertexNumInFacet := 1;
      FacetNum := FacetNum + 1; { bottom cap }
      LastFacet := LastFacet + 1;
      AddVertex( CosD(Theta+HalfDTheta),
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```

                                SinD(Theta+HalfDTheta), -1);
AddVertex( CosD(Theta-HalfDTheta),
                                SinD(Theta-HalfDTheta), -1);
AddVertex( CosD(180-Theta+HalfDTheta),
                                SinD(180-Theta+HalfDTheta), -1);
AddVertex( CosD(180-Theta-HalfDTheta),
                                SinD(180-Theta-HalfDTheta), -1);
VertexNumInFacet := 1;
Theta := Theta + DTheta
End
End;

Procedure MakeCylinderDatabase;

Begin
  For FacetNum := 1 To LastFacet Do
    Begin
      AddVertex( CosD(Theta-HalfDTheta),
                  SinD(Theta-HalfDTheta), -1 );
      AddVertex( CosD(Theta+HalfDTheta),
                  SinD(Theta+HalfDTheta), -1 );
      AddVertex( CosD(Theta+HalfDTheta),
                  SinD(Theta+HalfDTheta), 1 );
      AddVertex( CosD(Theta-HalfDTheta),
                  SinD(Theta-HalfDTheta), 1 );
      Theta := Theta + DTheta;
      VertexNumInFacet := 1
    End;
  EndCaps
End;

{
  {
    MAIN PROGRAM
  }
}
```



```

Begin
  InitVertexMaker;

  SetupCylinder(90);
  { Num Facets => 360 Div Even Number = Integer (90 is max) }

  MakeCylinderDatabase;

  SaveData('Cylinder.Dat');

  ExitGraphics
End.

```

Figure 9-4. Listing of Program to Generate Cylinders

Generating the Data File for a Hemisphere

The program *HmSphere.Pas* generates the object data file for a hemisphere. This program is listed in Figure 9-5. The program calls *InitVertexMaker* to do the initialization necessary before making an object data file. It calls the procedure *SetupHemiSphere*. Two parameters pass to this procedure. The first is the number of facets in a horizontal row projected onto the face of the hemisphere, and the second is the number of facets in a vertical row. This program sets these values to 60 and 45 respectively, which are the maximum allowable values. The procedure *SetupHemiSphere* sets starting values into the variables *VertexNum*, *VertexNumInFacet*, *Last Facet*, *LastVertexNumInFacet*, and *LastVertex* as in the program described above. It creates the parameters *DTheta* and *HalfDTheta* which represent angular increments horizontally around the base of the hemisphere. The parameter *Theta* is set to the starting angular position. The procedure creates the parameters *DPhi* and *HalfDPhi* which represent angular increments vertically on the hemisphere surface. The parameter *Phi* is set to the starting angular position. Getting back to the main program, the procedure *MakeSphereDataBase* is called next. This procedure begins by setting *Horz* and *Vert* to one. It continues with a *For* loop which iterates for each facet. At each iteration, the loop begins by computing the sine and cosine of *Phi* and scaling them, appropriately. It adds the four vertices for the facet to the vertex list. The vertex coordinates are a little too complicated to list here,

especially since new values are computed for the sine and cosine of *Phi* before the third and fourth vertices are calculated. However, the vertex coordinate locations should be clear from looking at the program listing. The procedure resets *VertexNumInFacet* to one, increments *Horz* by one, and increases *Theta* by *DTheta*. The value of *Horz* is compared with the maximum allowable value. If the maximum has been reached, *Horz* resets to one, *Vert* increments, *Theta* resets to its starting value, and *Phi* is increased by *DPhi*. The procedure loops again until all facets are processed. After the loop is complete, the procedure *EndCap* generates the cap at the base of the hemisphere. This procedure is similar to the procedure of the same name used in generating the pyramid and cone, except that the number of facets is determined from the number of horizontal facets rather than the entire number of facets in the sphere, and that the location of the end cap is at zero in the *z* plane rather than at -1 . After *EndCap* is complete, return to the main program, which calls *SaveData* to create the hemisphere object data file and save all of the vertex and facet data to it. The program returns to the text display mode and ends.

```
{
  

Hemisphere Database Generator


}
```

Uses

Dos, Crt;

{ \$I Math.Inc }

{ \$I Graph.Inc }

{ \$I Model.Inc }

{ \$I ShpMk.Inc }

CREATING OBJECT DATABASES

Var

```
Theta, DTheta, HalfDTheta : Integer;  
Phi, DPhi, HalfDPhi, Scale : Integer;  
HorzLoop, VertLoop      : Integer;  
Horz, Vert              : Integer;  
SinPhi, CosPhi          : Real;  
NumFacetsOnEndCap, T : Integer;
```

Procedure EndCap;

Begin

```
NumFacetsOnEndCap := HorzLoop Div 2 - 1;
```

```
If NumFacetsOnEndCap And 1 = 1 Then
```

```
    Theta := -90 + HalfDTheta
```

```
Else
```

```
    Theta := -90 + DTheta;
```

```
For T := 1 To NumFacetsOnEndCap Do { bottom cap }
```

```
    Begin
```

```
        FacetNum := FacetNum + 1;
```

```
        LastFacet := LastFacet + 1;
```

```
        AddVertex( CosD(Theta+HalfDTheta),
```

```
                    SinD(Theta+HalfDTheta), 0 );
```

```
        AddVertex( CosD(Theta-HalfDTheta),
```

```
                    SinD(Theta-HalfDTheta), 0 );
```

```
        AddVertex( CosD(180-Theta+HalfDTheta),
```

```
                    SinD(180-Theta+HalfDTheta), 0 );
```

```
        AddVertex( CosD(180-Theta-HalfDTheta),
```

```
                    SinD(180-Theta-HalfDTheta), 0 );
```

```
        VertexNumInFacet := 1;
```

```
        Theta := Theta + DTheta
```

```
    End
```

```
End;
```

Procedure SetupHemisphere(Horizontal, Vertical : Integer);

Begin

```
VertexNum := 1;
```

```
VertexNumInFacet := 1;
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
HorzLoop := Horizontal;  
VertLoop := Vertical;  
  
LastFacet := HorzLoop * VertLoop;  
LastVertexNumInFacet := MaxVertexNumInFacet;  
LastVertex := LastFacet * LastVertexNumInFacet;  
DTheta := 360 Div HorzLoop;  
HalfDTheta := DTheta Div 2;  
Theta := HalfDTheta;  
  
DPhi := 90 Div VertLoop;  
HalfDPhi := DPhi Div 2;  
Phi := HalfDPhi;  
  
Scale := 1  
End;  
  
Procedure MakeSphereDatabase;  
  
Begin  
  Horz := 1;  
  Vert := 1;  
  For FacetNum := 1 To LastFacet Do  
    Begin  
      SinPhi := Scale * SinD(Pi + HalfDPhi);  
      CosPhi := Scale * CosD(Pi + HalfDPhi);  
      AddVertex( SinPhi * CosD(Theta-HalfDTheta),  
                SinPhi * SinD(Theta-HalfDTheta), CosPhi );  
      AddVertex( SinPhi * CosD(Theta+HalfDTheta),  
                SinPhi * SinD(Theta+HalfDTheta), CosPhi );  
      SinPhi := Scale * SinD(Pi - HalfDPhi);  
      CosPhi := Scale * CosD(Pi - HalfDPhi);  
      AddVertex( SinPhi * CosD(Theta+HalfDTheta),  
                SinPhi * SinD(Theta+HalfDTheta), CosPhi );  
      AddVertex( SinPhi * CosD(Theta-HalfDTheta),  
                SinPhi * SinD(Theta-HalfDTheta), CosPhi );  
      VertexNumInFacet := 1;  
      Horz := Horz + 1;  
      Theta := Theta + DTheta;  
      If Horz > HorzLoop Then
```


CREATING OBJECT DATABASES

```
Begin
    Horz := 1;
    Vert := Vert + 1;
    Theta := HalfDTheta;
    Phi := Phi + DPhi
End
End;
EndCap
End;
{
    MAIN PROGRAM
}
Begin
    InitVertexMaker;

    SetupHemisphere(60, 45);
    { Horz, Vert => 360 Div Horz, 90 Div Vert = Even Integer
      (60,45 is max) }

    MakeSphereDatabase;

    SaveData('HmSphere.Dat');

    ExitGraphics
End.
```

Figure 9-5. Listing of Program to Generate Hemispheres

Generating the Data File for a Sphere

The program *Sphere.Pas* generates the object data file for a sphere. This program is listed in Figure 9-6. The program begins by calling *InitVertexMaker* to do the necessary initialization before making an object data file. It calls *SetupSphere*. Two

parameters pass to this procedure. The first is the number of facets in a horizontal row projected onto the face of the hemisphere, and the second is the number of facets in a vertical row. This program sets these values to 60 and 45 respectively, which are the maximum allowable values. *SetupSphere* first sets starting values into the variables *VertexNum*, *VertexNumInFacet*, *Last Facet*, *LastVertexNumInFacet*, and *LastVertex* as described above. It creates the parameters *DTheta* and *HalfDTheta* which represent angular increments horizontally around the perimeter of the sphere. The parameter *Theta* sets to the starting angular position. The procedure creates parameters *DPhi* and *HalfDPhi*, which represent angular increments vertically on the sphere surface. The parameter *Phi* is set to the starting angular position. The only difference between this procedure and the corresponding one for the hemisphere, is that the vertical facets are made twice as large, so they cover the full surface of the sphere vertically, instead of half the vertical surface. Getting back to the main program, the procedure *MakeSphereDataBase* is called. This procedure is the same as the procedure of the same name used in generating the hemisphere. When it is complete, return to the main program, which calls *SaveData* to create the sphere object data file and save all of the vertex and facet data to it. The program returns to the text display mode and ends.

```
{
  

Sphere Database Generator


}
```

Uses

Dos, Crt;

{\$I Math.Inc}

{\$I Graph.Inc}

{\$I Model.Inc}

{\$I ShpMk.Inc}

CREATING OBJECT DATABASES

Var

```
Theta, DTheta, HalfDTheta : Integer;  
Phi, DPhi, HalfDPhi : Integer;  
HorzLoop, VertLoop : Integer;  
Horz, Vert : Integer;  
SinPhi, CosPhi : Real;
```

Procedure SetupSphere(Horizontal, Vertical : Integer);

Begin

```
VertexNum := 1;  
VertexNumInFacet := 1;
```

```
HorzLoop := Horizontal;  
VertLoop := Vertical;
```

```
LastFacet := HorzLoop * VertLoop;  
LastVertexNumInFacet := MaxVertexNumInFacet;  
LastVertex := LastFacet * LastVertexNumInFacet;
```

```
DTheta := 360 Div HorzLoop;  
HalfDTheta := DTheta Div 2;  
Theta := HalfDTheta;  
DPhi := 180 Div VertLoop;  
HalfDPhi := DPhi Div 2;  
Phi := HalfDPhi
```

End;

Procedure MakeSphereDatabase;

Begin

```
Horz := 1;  
Vert := 1;
```

```
For FacetNum := 1 To LastFacet Do
```

Begin

```
SinPhi := SinD(Phi + HalfDPhi);  
CosPhi := CosD(Phi + HalfDPhi);  
AddVertex( SinPhi * CosD(Theta-HalfDTheta),  
           SinPhi * SinD(Theta-HalfDTheta), CosPhi );
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
        AddVertex( SinPhi * CosD(Theta+HalfDTheta),
                   SinPhi * SinD(Theta+HalfDTheta), CosPhi );
        SinPhi := SinD(Phi - HalfDPhi);
        CosPhi := CosD(Phi - HalfDPhi);
        AddVertex( SinPhi * CosD(Theta+HalfDTheta),
                   SinPhi * SinD(Theta+HalfDTheta), CosPhi );
        AddVertex( SinPhi * CosD(Theta-HalfDTheta),
                   SinPhi * SinD(Theta-HalfDTheta), CosPhi );
        VertexNumInFacet := 1;
        Horz := Horz + 1;
        Theta := Theta + DTheta;
        If Horz > HorzLoop Then
            Begin
                Horz := 1;
                Vert := Vert + 1;
                Theta := HalfDTheta;
                Phi := Phi + DPhi
            End
        End
    End;

{
    MAIN PROGRAM
}

Begin
    InitVertexMaker;

    SetupSphere(60, 45);
    { Horz, Vert => 360 Div Horz, 180 Div Vert = Even Integer
      (60,45 is max) }

    MakeSphereDatabase;

    SaveData('Sphere.Dat');

    ExitGraphics
End.
```

Figure 9-6. Program to Generate Spheres

Generating Data Files for Equations

The program *PlotEqns.Pas* generates the object data files for a grid and four different equations. Once you become familiar with the method, it will be easy to adapt the program for other equations. The program is listed in Figure 9-7. At the beginning of the program, the values for the parameters *ObjF* (which is the name of the object file), *Span*, *Contour*, and *Offset* (for the grid and four different equations), are listed. In addition, each set of values is accompanied by an instance of the function *zf*, which determines the value of *z* as a function of *x* and *y* for the particular case. For each run of the program, all of these sets of values, except one, are commented out.

The program calls *InitVertexMaker* to do the necessary initialization before making an object data file. It calls the procedure *SetupPlotEqn*, passing it the parameters for the left and right *x* boundaries (which are set to $-Span$ and $Span$), the bottom and top *y* boundaries (which are set to $-Span$ and $Span$), the number of facets in the horizontal direction (which is set to *Contour*), and the number of facets in the vertical direction (which is set to *Contour*). *SetupPlotEqn* first sets starting values into the variables *VertexNum*, *VertexNumInFacet*, *LastFacet*, *LastVertexNumInFacet*, and *LastVertex* in the same way that these parameters were set up in the program described above. The procedure creates scaling factors *sx*, *sy*, and *sz*. It creates the parameters *Dx* and *HalfDx* which represent increments in the horizontal direction, and *Dy* and *HalfDy* which represent increments in the vertical direction. The limits *Xlft*, *Ytop*, *Xrgt*, and *Ybot*, which were passed to this procedure, are stored in global parameters *ix*, *iy*, *ex*, and *ey*, respectively, completing the action of the procedure.

```
{
  Equation Plot and Grid Database Generator
}
```

```
Uses
  Dos, Crt;
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
{ $I Math.Inc }
```

```
{ $I Graph.Inc }
```

```
{ $I Model.Inc }
```

```
{ $I ShpMk.Inc }
```

```
{
```

Equations

```
}
```

```
{ Const
```

```
  ObjF = 'Grid.Dat';
```

```
  Span = 5;
```

```
  Contour = 14;
```

```
  Offset = 0.5;
```

```
Function zf(x, y : Real) : Real;
```

```
  Begin
```

```
    zf := 0
```

```
  End;
```

```
}
```

```
{
```

```
Const
```

```
  ObjF = 'PlotEqn1.Dat';
```

```
  Span = 5;
```

```
  Contour = 58;
```

```
  Offset = 0;
```

```
Function zf(x, y : Real) : Real;
```

```
  Var
```

```
    c : Real;
```


CREATING OBJECT DATABASES

```
Begin
  c := Sqr(x) + Sqr(y);
  zf := 75 / ( c + 1 )
End;
}
```

```
{
Const
  ObjF = 'PlotEqn2.Dat';
  Span = 5;
  Contour = 58;
  Offset = 0;

Function zf(x, y : Real) : Real;
```

```
  Begin
    zf := 6 * (Cos(x * y) + 1);
  End;
}
```

```
{
Const
  ObjF = 'PlotEqn3.Dat';
  Span = 5;
  Contour = 58;
  Offset = 0;

Function zf(x, y : Real) : Real;
```

```
  Var
    c : Real;

  Begin
    c := Sqr(x) + Sqr(y);
    zf := 20 * ( Sin( Sqrt(c) ) + 1 )
  End;
}
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

Const

ObjF = 'PlotEqn4.Dat';

Span = 2.75;

Contour = 58;

Offset = 0;

Function zf(x, y : Real) : Real;

Var

c : Real;

Begin

c := Sqr(x) + Sqr(y);

zf := 10 * (1 + Sqrt(c) + Sin(Sqr(c)*0.1) + Sin(x*y));

End;

Var

Dx, HalfDx : Real;

Dy, HalfDy : Real;

x, y, z : Real;

sx, sy, sz : Real;

ix, iy : Real;

ex, ey : Real;

Procedure SetupPlotEqn(Xlft, Xrgt, Ybot, Ytop : Real;
HorzContours, VertContours : Integer; Offset : Real);

Begin

VertexNum := 1;

VertexNumInFacet := 1;

LastFacet := HorzContours * VertContours;

LastVertexNumInFacet := MaxVertexNumInFacet;

LastVertex := LastFacet * LastVertexNumInFacet;

sx := 200 / 100 / (Xrgt - Xlft) * 10 / 11;

sy := 200 / 100 / (Ytop - Ybot) * 10 / 11;

sz := 1 / 100;

Dx := (Xrgt - Xlft) / (HorzContours-1);

Dy := (Ybot - Ytop) / (VertContours-1);

CREATING OBJECT DATABASES

```
HalfDx := Dx / 2 / (Offset + 1);
HalfDy := Dy / 2 / (Offset + 1);
ix := Xlft;
iy := Ytop;
ex := Xrgt;
ey := Ybot
End;
```

Procedure MakePlotEqnDatabase; { make sure that an equation was chosen above }

Begin

FacetNum := 0;

X := ix;

Repeat

Y := iy;

Repeat

FacetNum := FacetNum + 1;

z := zf((x-HalfDx), (y+HalfDy));

AddVertex(sx * (x-HalfDx), sy * (y+HalfDy),
sz * z);

z := zf((x+HalfDx), (y+HalfDy));

AddVertex(sx * (x+HalfDx), sy * (y+HalfDy),
sz * z);

z := zf((x+HalfDx), (y-HalfDy));

AddVertex(sx * (x+HalfDx), sy * (y-HalfDy),
sz * z);

z := zf((x-HalfDx), (y-HalfDy));

AddVertex(sx * (x-HalfDx), sy * (y-HalfDy),
sz * z);

VertexNumInFacet := 1;

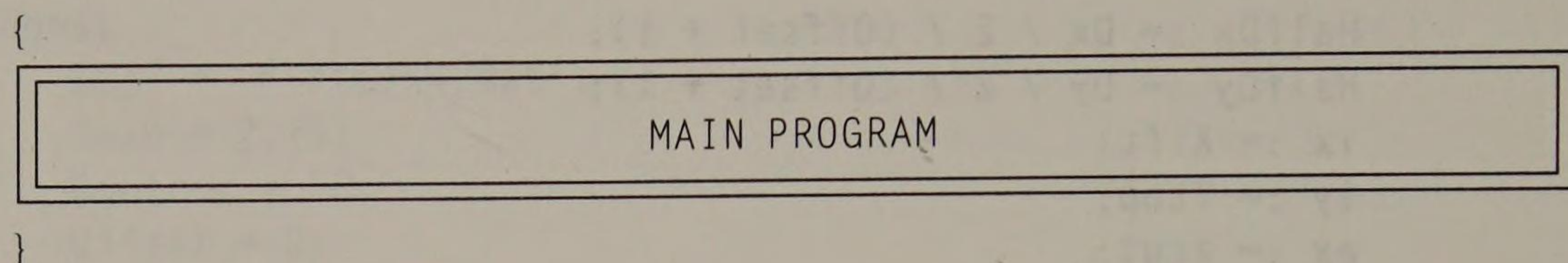
Y := Y + Dy

Until (Y < ey);

X := X + Dx

Until(X > ex)

End;



```

Begin
  InitVertexMaker;
  { Xlft, Xrgt, Ybot, Ytop, (X, Y) Contours , Offset }
  SetupPlotEqn(-Span, Span, -Span, Span, Contour, Contour,
               Offset);

  MakePlotEqnDatabase;
  SaveData(ObjF);
  ExitGraphics
End.

```

Figure 9-7. Program To Create Equation Databases

Getting back to the main program, the procedure *MakePlotEqnDatabase* is called next. It begins by setting X to ix and Y to iy , and continues with a pair of nested *Repeat* loops which iterate until every value of X and Y is treated, thus covering all facets of the displayed object. At each iteration, the value of z is computed for each of the four vertices of the facet, and the vertex is added to the vertex list. The coordinates for each of the four vertices of a facet are clear from the program listing. The inner loop resets *VertexNumInFacet* to one and increases the value of Y for each iteration. When the maximum value is reached, the outer loop increases the value of X and resets Y . After all vertices have been treated, you return to the main program, which calls *SaveData* to create the hemisphere object data file and save all of the vertex and facet data to it. The program returns to the text display mode and ends.

Generating the Data File for a Toroid

The program *Toroid.Pas* generates the object data file for a toroid. This program is listed in Figure 9-8. The program begins by calling *InitVertexMaker* to do the necessary initialization before making an object data file. It calls the procedure *SetupToroid*. Two parameters pass to this procedure. The first is the number of facets in a horizontal row projected onto the face of the toroid and the second is the number of facets in a vertical row. This program sets these values to 60 and 60, respectively,

which are the maximum allowable values. The procedure *SetupToroid* first sets starting values into the variables *VertexNum*, *VertexNumInFacet*, *Last Facet*, *LastVertexNumInFacet*, and *LastVertex* in the same way these parameters were described previously. It also sets the parameter *HorzLoop* and *VertLoop* to the maximum numbers of horizontal and vertical facets, respectively. It creates the parameters *DTheta* and *HalfDTheta*, which represent angular increments horizontally around the base of the toroid. The parameter *Theta* is set to the starting angular position. The procedure creates the parameters *DPhi* and *HalfDPhi* which represent vertical increments vertically on the toroid surface. The parameter *Phi* sets to the starting angular position.

Getting back to the main program, the procedure *MakeToroidDataBase* is called next. This procedure uses a nested pair of *Repeat* loops to generate the vertices for every one of the facets that make up the toroid. The outer loop sets *Phi* to its starting value and computes necessary sines and cosines for the current value of *Theta*. The inner loop computes the necessary sines and cosines of *Phi* multiplied by the radius. It computes the *x*, *y*, and *z* coordinates for each of the four vertices that make up the facet and adds each vertex to the vertex list. It increases *Phi* by *DPhi* and iterates again until it reaches the maximum value of the angle, whereupon it returns to the outer loop. At the end of each iteration of the outer loop, *Theta* increases by *DTheta* until the maximum value is reached, whereupon the loop and the procedure terminate. Now return to the main program, which calls *SaveData* to create the hemisphere object data file and save all of the vertex and facet data to it. The program returns to the text display mode and ends.

```
{
  Toroid Database Generator
}
```

Uses

Dos, Crt;

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
{ $I Math.Inc }
```

```
{ $I Graph.Inc }
```

```
{ $I Model.Inc }
```

```
{ $I ShpMk.Inc }
```

```
Var
```

```
  Theta, DTheta, HalfDTheta : Integer;
```

```
  Phi, DPhi, HalfDPhi : Integer;
```

```
  HorzLoop, VertLoop : Integer;
```

```
Procedure SetupToroid(Horizontal, Vertical : Integer);
```

```
  Begin
```

```
    VertexNum := 1;
```

```
    VertexNumInFacet := 1;
```

```
    HorzLoop := Horizontal;
```

```
    VertLoop := Vertical;
```

```
    LastFacet := HorzLoop * VertLoop;
```

```
    LastVertexNumInFacet := MaxVertexNumInFacet;
```

```
    LastVertex := LastFacet * LastVertexNumInFacet;
```

```
    DTheta := 360 Div HorzLoop;
```

```
    HalfDTheta := DTheta Div 2;
```

```
    Theta := HalfDTheta + 45 + 45; { offset for best display }
```

```
    DPhi := 360 Div VertLoop;
```

```
    HalfDPhi := DPhi Div 2;
```

```
    Phi := HalfDPhi
```

```
  End;
```

```
Procedure MakeToroidDatabase( Rho, R : Real );
```

```
  Var
```

```
    CosThetaMinus, SinThetaMinus : Real;
```

```
    CosThetaPlus, SinThetaPlus : Real;
```


CREATING OBJECT DATABASES

```
RCosPhiMinus , RSinPhiMinus : Real;  
RCosPhiPlus , RSinPhiPlus : Real;  
x, y, z : Real;
```

```
Begin
```

```
FacetNum := 0;
```

```
Repeat
```

```
Phi := HalfDPhi;
```

```
CosThetaMinus := CosD( Theta - HalfDTheta );
```

```
SinThetaMinus := SinD( Theta - HalfDTheta );
```

```
CosThetaPlus := CosD( Theta + HalfDTheta );
```

```
SinThetaPlus := SinD( Theta + HalfDTheta );
```

```
Repeat
```

```
RCosPhiMinus := R * CosD( Phi - HalfDPhi );
```

```
RSinPhiMinus := R * SinD( Phi - HalfDPhi );
```

```
RCosPhiPlus := R * CosD( Phi + HalfDPhi );
```

```
RSinPhiPlus := R * SinD( Phi + HalfDPhi );
```

```
FacetNum := FacetNum + 1;
```

```
x := (Rho + RCosPhiMinus) * CosThetaMinus;
```

```
y := (Rho + RCosPhiMinus) * SinThetaMinus;
```

```
z := RSinPhiMinus;
```

```
AddVertex( x, y, z );
```

```
x := (Rho + RCosPhiMinus) * CosThetaPlus;
```

```
y := (Rho + RCosPhiMinus) * SinThetaPlus;
```

```
z := RSinPhiMinus;
```

```
AddVertex( x, y, z );
```

```
x := (Rho + RCosPhiPlus) * CosThetaPlus;
```

```
y := (Rho + RCosPhiPlus) * SinThetaPlus;
```

```
z := RSinPhiPlus;
```

```
AddVertex( x, y, z );
```

```
x := (Rho + RCosPhiPlus) * CosThetaMinus;
```

```
y := (Rho + RCosPhiPlus) * SinThetaMinus;
```

```
z := RSinPhiPlus;
```

```
AddVertex( x, y, z );
```

```
VertexNumInFacet := 1;
```

```
Phi := Phi + DPhi
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
        Until Phi >= 360 + HalfDPhi;
        Theta := Theta + DTheta
    Until Theta >= 360 + HalfDTheta + 90 { offset for best
                                          display }
End;

{


MAIN PROGRAM


}
Begin
    InitVertexMaker;

    SetupToroid(60,60);
        { Horz, Vert => 360 Div Horz, 360 Div Vert = Even Integer
          (60,60 is max) }

    MakeToroidDatabase(0.75, 0.25); { toroid radius, circle
                                     radius }

    SaveData('Toroid.Dat');

    ExitGraphics
End.
```

Figure 9-8. Program to Generate Toroids

Generating Data Files for Solids of Revolution

The program *SolOfRev.Pas* generates the object data files for a number of solids of revolution. This program is listed in Figure 9-9. Suppose you have a two-dimensional figure symmetrical about an axis and you rotate this figure about that axis. The surface produced by the outline of that figure during rotation is a solid of revolution. All sorts of interesting figures are produced by this technique. The *SolOfRev.Pas* program provides a technique for generating such solids as well as providing the data to generate a number of interesting object data files. Figure 9-10 shows a two-dimensional shape and the three-dimensional result of rotating it about the *z* axis. The program begins by calling *InitVertexMaker* to do the necessary ini-

tialization before making an object data file. It then calls *SetupSolidOfRevolution* and two parameters pass to it. The first is the number of facets in a horizontal row around the circumference of the solid, and the second is the number of facets in a vertical row. This program sets these values to 60 and 60, respectively, which are the maximum allowable values. The procedure *SetupSolidOfRevolution* first sets starting values into the variables *VertexNum*, *VertexNumInFacet*, *Last Facet*, *LastVertexNumInFacet*, and *LastVertex* as described above. It also sets the parameters *HorzLoop* and *VertLoop* to the maximum numbers of horizontal and vertical facets, respectively, that were passed to it. It creates the parameters *DTheta* and *HalfDTheta*, which represent angular increments horizontally around the base of the toroid. The parameter *Theta* is set to the starting angular position. The procedure creates the parameters *DZ* and *HalfDZ*, which represent vertical steps on the solid surface. Getting back to the main program, the procedure *MakeSolidOfRevolution* is called next. This procedure first clears *RS* which contains 60 values that determine the shape of the two-dimensional figure to be rotated. Next, the procedure *RadialArray* is called. This procedure loads the array *RS* with the information needed to describe the two-dimensional object, and also sets up the name for the object data file. A number of versions of this procedure are given at the beginning of the listing, each describing a differently shaped object. All but one are commented out; if you wish to create one particular object file, make sure that its version of this procedure is the only one *not* commented out. Alternately, you can think up your own version of the procedure to create a new shape. Once the procedure *RadialArray* is complete, the procedure *DisplayRadialArray* is called. This procedure displays a silhouette of the desired solid in the lower right corner of the screen. Next, *MakeSolidOfRevolution* uses a nested pair of *Repeat* loops to generate the vertices for every one of the facets that make up the solid of revolution. The outer loop sets *I* to one and computes necessary sines and cosines for the current value of *Theta*, and the current value of *Zs*. The inner loop computes the necessary values for the *z* coordinate. It computes the *x*, *y*, and *z* coordinates for each of the four vertices that make up the facet and adds each vertex to the vertex list. It sets *VertexNumberInFacet* to one and increments *I*. When it has cycled through all values of *I*, it returns to the outer loop. At the end of each iteration of the outer loop, *Theta* increases by *DTheta* until the maximum value

is reached, whereupon the loop and the procedure terminate. Now return to the main program, which calls *SaveData* to create the hemisphere object data file and save all of the vertex and facet data to it. The program returns to the text display mode and ends.

```
{
  Solid of Revolution Database Generator
}
```

Uses

Dos, Crt;

{\$I Math.Inc}

{\$I Graph.Inc}

{\$I Model.Inc}

{\$I ShpMk.Inc}

{

Radial Arrays

}

Var

I : Integer;

RS : Array[1..60] Of Real;

{

CREATING OBJECT DATABASES

```
Const
  ObjF = 'LgBeads.Dat';

Procedure RadialArray; (* large beads *)

Begin
  For I := 1 To 59 Do
    RS[I] := SinD(I*10) * 0.5 + 0.5;
  RS[60] := 0.05
End;
}

{
Const
  ObjF = 'Beads.Dat';

Procedure RadialArray; (* string of beads *)

Begin
  For I := 1 To 60 Do
    RS[I] := SinD((I-1)*9) * 0.5
  End;
}

{
Const
  ObjF = 'Funnels.Dat';

Procedure RadialArray; (* funnels *)

Begin
  For I := 1 To 60 Do
    RS[I] := Sin( Sqrt(60.1 - I) - 0.5 ) * 0.45 + 0.5
  End;
}

{
Const
  ObjF = 'MechPart.Dat';

Procedure RadialArray; (* mechanical part *)
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
Begin
  For I := 1 To 6 Do
    RS[I] := I / 6 * 0.5 + 0.1;
  For I := 7 To 20 Do
    RS[I] := 0.6;
  For I := 21 To 32 Do
    RS[I] := 0.6 + (I-21) / 32;
  For I := 33 To 53 Do
    RS[I] := 1.0;
  For I := 54 To 60 Do
    RS[I] := 0.1
End;
}

{
Const
  ObjF = 'Rocket.Dat';

Procedure RadialArray; (* rocket ship *)

Begin
  For I := 60 DownTo 53 Do
    RS[I] := 0.02;
  For I := 52 DownTo 47 Do
    RS[I] := (52-I) / 52 + 0.05;
  For I := 46 DownTo 35 Do
    RS[I] := 0.16;
  For I := 34 DownTo 27 Do
    RS[I] := (34-I) / 34 + 0.19;
  For I := 26 DownTo 5 Do
    RS[I] := 0.4;
  For I := 4 DownTo 1 Do
    RS[I] := (4-I) / 22 + 0.4
End;
}

Const
  ObjF = 'ChesPawn.Dat';
```


CREATING OBJECT DATABASES

```
Procedure RadialArray; (* chess pawn *)
```

```
Begin
```

```
  For I := 60 DownTo 44 Do
```

```
    RS[I] := SinD( (60-I) * 9 ) * 0.6;
```

```
  For I := 43 DownTo 11 Do
```

```
    RS[I] := 0.2;
```

```
  For I := 10 DownTo 1 Do
```

```
    RS[I] := 0.2 + SinD( (10-I) * 12 ) * 0.5
```

```
End;
```

```
Procedure DisplayRadialArray;
```

```
Begin
```

```
  For I := 1 To 60 Do
```

```
    Draw(260 - Round(60 * RS[I]), 150 - I,  
         260 + Round(60 * RS[I]), 150 - I, 35)
```

```
End;
```

```
Var
```

```
  Theta, DTheta, HalfDTheta : Integer;
```

```
  Zs, DZ, HalfDZ      : Real;
```

```
  HorzLoop, VertLoop   : Integer;
```

```
Procedure SetupSolidOfRevolution(Horizontal, Vertical : Integer);
```

```
Begin
```

```
  VertexNum := 1;
```

```
  VertexNumInFacet := 1;
```

```
  HorzLoop := Horizontal;
```

```
  VertLoop := Vertical;
```

```
  LastFacet := HorzLoop * (VertLoop-1);
```

```
  LastVertexNumInFacet := MaxVertexNumInFacet;
```

```
  LastVertex := LastFacet * LastVertexNumInFacet;
```

```
  DTheta := 360 Div HorzLoop;
```

```
  HalfDTheta := DTheta Div 2;
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
Theta := HalfDTheta + 90; { offset for best display }

DZ := 2 / VertLoop;
HalfDZ := DZ / 2;
Zs := HalfDZ
End;
```

Procedure MakeSolidOfRevolutionDatabase;

```
Var
  CosThetaMinus, SinThetaMinus : Real;
  CosThetaPlus, SinThetaPlus : Real;
  ZMinus, ZPlus, x, y, z : Real;
Begin
  FacetNum := 0;

  For I := 1 To 60 Do { clear radial array }
    RS[I] := 0.0;

  RadialArray;

  DisplayRadialArray;

  Repeat

    I := 1;

    CosThetaMinus := CosD( Theta - HalfDTheta );
    SinThetaMinus := SinD( Theta - HalfDTheta );
    CosThetaPlus := CosD( Theta + HalfDTheta );
    SinThetaPlus := SinD( Theta + HalfDTheta );

    Zs := -1 + HalfDZ;

    Repeat

      ZMinus := Zs - HalfDZ;
      ZPlus := Zs + HalfDZ;

      FacetNum := FacetNum + 1;
```


CREATING OBJECT DATABASES

```
x := CosThetaMinus * RS[I];  
y := SinThetaMinus * RS[I];  
z := ZMinus;  
AddVertex( x, y, z );
```

```
x := CosThetaPlus * RS[I];  
y := SinThetaPlus * RS[I];  
z := ZMinus;  
AddVertex( x, y, z );
```

```
x := CosThetaPlus * RS[I+1];  
y := SinThetaPlus * RS[I+1];  
z := ZPlus;  
AddVertex( x, y, z );
```

```
x := CosThetaMinus * RS[I+1];  
y := SinThetaMinus * RS[I+1];  
z := ZPlus;  
AddVertex( x, y, z );
```

```
VertexNumInFacet := 1;
```

```
I := I + 1;
```

```
Zs := Zs + DZ
```

```
Until (I = 60);
```

```
Theta := Theta + DTheta
```

```
Until (Theta >= 360 + HalfDTheta + 90) { offset for best  
display }
```

```
End;
```

```
{
```

```
MAIN PROGRAM
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
}  
  
Begin  
  InitVertexMaker;  
  
  SetupSolidOfRevolution(60,60);  
    { Horz, Vert => 360 Div Horz, 360 Div Vert = Integer (60,60  
                                          is max) }  
  
  MakeSolidOfRevolutionDatabase;  
  
  SaveData(ObjF);  
  
  ExitGraphics  
End.
```

Figure 9-9. Program to Generate Solids of Revolution

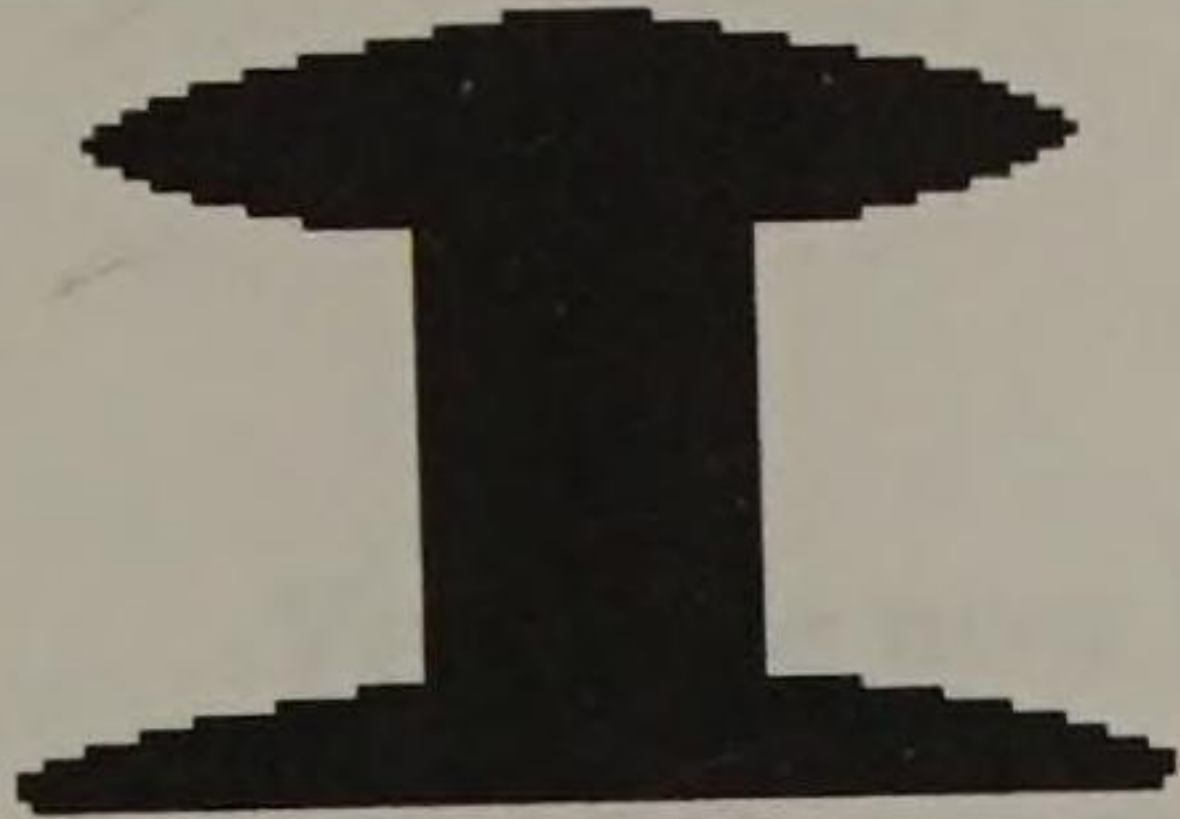


Figure 9-10. Profile of chess pawn used to generate solid of revolution.



Figure 9-11. Solid Rendering of Chess Pawn

Creating a Scene File

You have now learned how to create data files for generic objects and how to display solid-modeled scenes. Some typical scene data files are supplied on the program disk (listed in Appendix A), so you have probably already produced some interesting scenes on your VGA display. However, you haven't yet learned how to generate a scene file to give your artistic talents full rein. This chapter will describe how to generate your own scene files using the program *ScnMaker.Pas*, which is listed in Figure 10-2.

Using the *ScnMaker.Pas* Program

Before going into a detailed description of the *ScnMaker.Pas* program, use this section of Chapter 10 as a guide to the use of the program. As the program progresses, it displays various queries to which you must respond with entries. Although most of the information you enter is in previous sections, it isn't always easy to understand just what is meant, and what is needed when a question appears on the screen. The program sometimes doesn't help much. For example, if you are supposed to enter an integer and put in a floating-point number, instead, the program will quit with an error message and you have to start over. This section is intended to help you get a better understanding of how the program runs so that once you start working with it, your work should be error-free. It begins by displaying the following:

```
Program to Generate a Scene File for Model
```

```
Enter File Name =>
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

At this point, you need to enter up to an eight character file name acceptable to DOS. This will be the name of your scene file. Don't enter an extension. The program will automatically supply the extension *.SCN*. If you just hit the *Ent* key, instead of entering a file name, the program will name the file *Test*. Next, the program displays:

Description of the Scene =>

At this point you can type in a descriptive line for the scene, to be written to the disk file. This description identifies the context of the file, but is not actually used for anything. Next, the program displays:

Perspective Display?

You will normally want perspective in your scene, since it makes the scene more realistic by causing near objects to appear larger than far ones. If you decide you want perspective, answer *Y*; otherwise answer *N*. If you selected perspective, the program will next ask you:

(Mx My Mz D) =

You now need to enter four integers, separated by spaces. The first three integers are the *x*, *y*, and *z* coordinates of the observer position, and the fourth integer is the distance from the observer to the screen. The values (0 0 500 500) are a good place to start. The display will then show:

Direction for Viewing

Around the z-Axis =

This is the first of two angles that define the direction the viewer is looking (which is not necessarily toward the center of the scene). An integer should be entered. Try 245 as a starting value. Next the display will show:

Off of the z-Axis =

This is the second viewing angle. Again, an integer should be entered. A good trial value is 25. The display will then ask:

Direction of Light

CREATING A SCENE FILE

Around the z-Axis =

This is the first of two angles determining the direction of the light source that illuminates the scene. The position of the light source determines how the scene is lighted. Again an integer should be entered. A good trial value is 45. The display will then show:

Off the z-Axis =

This is the second light source angle. Again an integer should be entered. A good trial value is 45. You will remember that the angles are as defined in Figure 10-1.

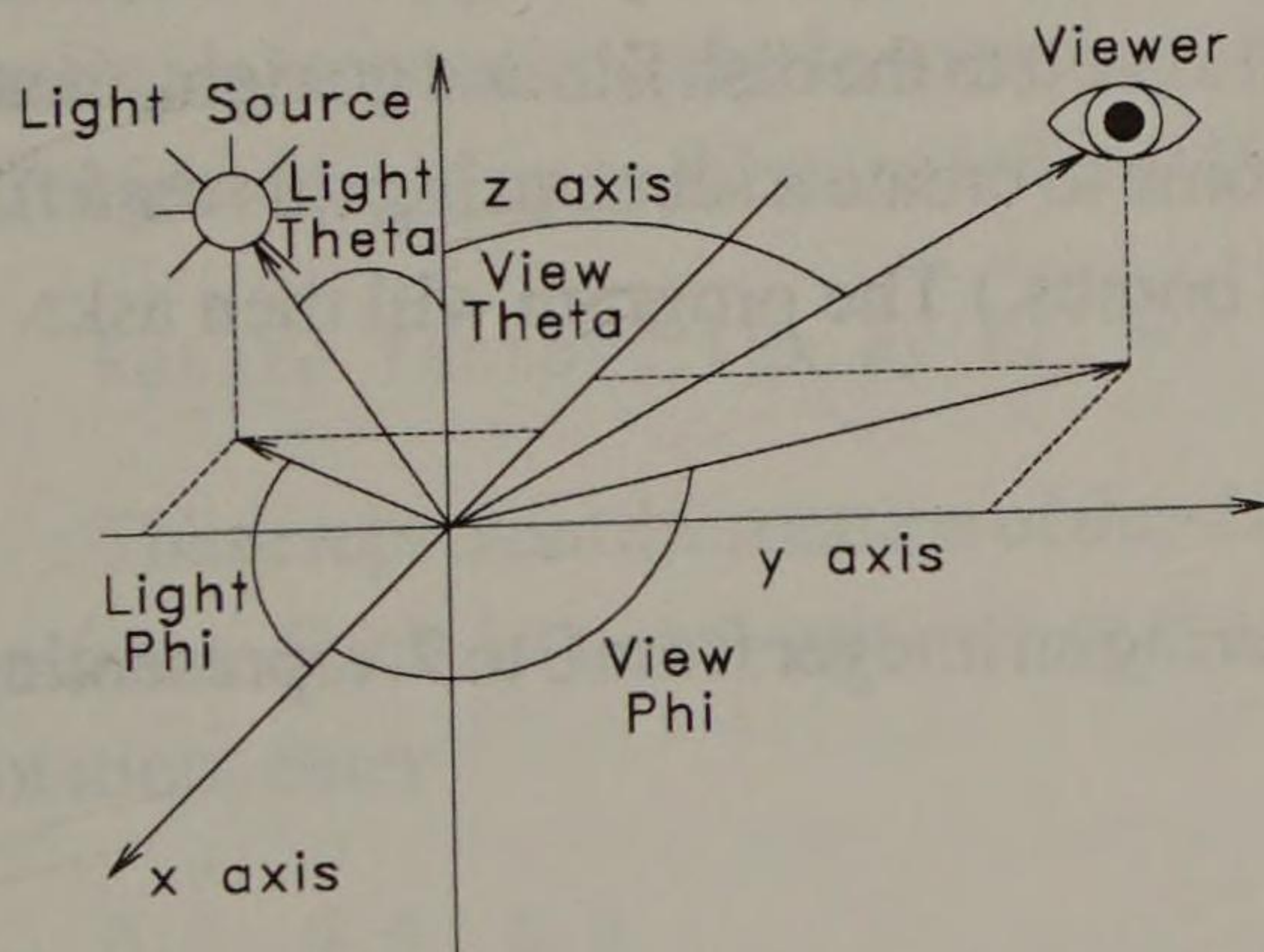


Figure 10-1. Viewer and Light Source Angles

Next, the program will ask:

Vertical Sort of Objects?

If you want the objects sorted so nearer objects are written over farther objects, answer *Y*; otherwise answer *N*. The display will then show:

Place edge Reflectors at (x=-100, y=-100, and z=-100)?

If you respond *Y*, three walls consisting of mirror tiles will be generated at the coordinates shown. If you don't want these walls, respond *N*. The display will then show:

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

Place edge Reflectors at (x=-100, y=-100, and z=0)?

If you respond *Y*, three walls consisting of mirror tiles will be generated at the coordinates shown. If you don't want these walls, respond *N*. If you should happen to respond with a *Y* to both questions, the second response will override the first when the file is read and the walls will appear at the second location. The display will next show:

Now add objects to the scene ->
Enter the Objects Name =>

At this point, type in the name of an object data file, without the extension. (Note that if you type in the name of the object file incorrectly or if you type in the name of a non-existent object file, the program will save it to the disk file and go right ahead as if nothing is wrong. It is only when you come to create a scene using this data file that you get an error message and program bombs.) The program will then ask:

Color Of the Object?

You need to respond to this query by entering an integer from 0 to 7, representing colors as shown in Table 10-1.

Number	Color
0	Black
1	Blue
2	Green
3	Cyan
4	Red
5	Magenta
6	Brown
7	Gray

Table 10-1. Color Assignments

The 256 color palette is set up so it contains 35 shades of each of the colors from 1 to 7 as well as black. The color you select for the object is the basic color; which shade is chosen for each facet depends on the position and lighting of the facet. Next the display will show:

Scale Factors (Sx Sy Sz) =>

Each object is normally sized at about 1 x 1 x 1. This is a small object on the screen. You can scale object to any size you desire, with independent scalings for each of the three coordinates. Do this by entering scaling factors for x, y, and z. These may be real numbers having up to four places for the non-decimal part of the number, and up to four decimal places. The three numbers should be separated by spaces. The scaling is, approximately, in screen coordinates. The screen is 320-by-200 pixels, so an object scaled up to this size will fill the entire screen. The display will then show:

Rotate factors (Rx Ry Rz) =>

These represent the rotation of the object about the x, y, and z axes from its nominal position. Each is a real number and an angle in degrees. If you don't want any rotation, enter

0.0 0.0 0.0

The display will next show:

Translate Factors (Tx Ty Tz) =>

These enable you to specify moving the object from its nominal position at the center of the screen. You may enter three real numbers, separated by spaces, which represent translation amounts in the x, y, and z directions. Remember that the screen is 320-by-200 pixels. If you translate farther than this, the object will likely be off the screen, altogether. The display will now ask:

Reflect this Object in Mirrored Objects?

You will normally want to answer Y to this, since an object that doesn't reflect

in mirrors is a very peculiar object, indeed. The display will then ask:

Allow this Object to be Sorted for Placement?

Again, the usual answer is *Y* unless you are creating background walls or floors. The display will then ask:

This Object is a Mirror?

Answer *Y* or *N* according to your desires. The display will then ask:

Add another Object to Scene?

If you answer *Y* to this, the program loops through all of the queries about an object, again, giving you the opportunity to add another object to the scene. If you answer *N*, the program will terminate, saving the scene file.

Keyboard Response Routines

Before describing the main program, look at the functions to obtain a response from the keyboard. First let's look at the function *Response*. This is the same as the function of the same name in Chapter 7. This function obtains a response to a question and makes sure it is a *Y* or an *N*. It does this with two, nested *Repeat* loops. The inner loop keeps trying to read a key from the keyboard until the response is other than a null, indicating that a key has been pressed. Then the character received from the keyboard converts to a capital letter. If it is a *Y* or an *N*, the outer loop terminates; otherwise the loops continue to iterate until an acceptable response is received. Note that whenever *Response* is called in a program, nothing happens until you type either *Y*, *y*, *N*, or *n*. When a correct response is received, if it is *Y*, the function returns *True*; otherwise, it returns *False*.

Next, let's look at the function *NumberResponse*. This is basically the same in operation as above. The principal difference is that instead of demanding a response of *Y*, *y*, *N*, or *n*, the function requires pressing one of the number keys from 0 to 9. When this occurs, the function converts the ASCII character received from the keyboard to a number from 0 to 9 and returns this number.

The *ScnMaker.Pas* Program

The *ScnMaker.Pas* program clears the screen and displays a title. It calls the procedure *GetSceneFile*, which asks for the file name to be entered, and stores the cursor position. If the *Ent* key is hit, it names the file *Test*, returns to the stored cursor position and writes *Test* to the screen. It then returns to the main program, which calls the procedure *SaveScene*. This procedure appends the extension *.SCN* to the file name. It opens a text file having the assigned name for writing. Next, the procedure writes out to the display, asking for a description of the scene. It reads in a line of text and writes it out to the disk file. It asks if a perspective display is desired, then writes the response from the keyboard to the disk file. If the response was *Y* (Boolean *True*), the procedure asks for the parameters *Mx*, *My*, *Mz*, and *D*. When these are entered, in the form of three digit integers separated by spaces and followed by the *Ent* key, they are written to the disk file. If perspective was not selected, four zeroes are written to the disk file instead of asking for these parameters. Next, the procedure asks for the viewing direction around the *z* axis. It reads the keyboard response and writes it to the disk file. Next, the procedure asks for the viewing direction off of the *z* axis. It reads the keyboard response and writes it to the disk file. Next, the procedure asks for the light direction around the *z* axis. It reads the keyboard response and writes it to the disk file. Next, the procedure asks for the viewing direction off of the *z* axis. It reads the keyboard response and writes it to the disk file. The procedure asks if a vertical sort of objects is to be made. It writes the Boolean result obtained from *Response* to the disk file. The procedure asks if edge reflectors are to be placed at $(-100, -100, -100)$. It writes the Boolean result obtained from *Response* to the disk file. The procedure asks if edge reflectors are to be placed at $(-100, -100, 0)$. It writes the Boolean result obtained from *Response* to the disk file. The procedure writes to the display, *Now add objects to the scene ->*. It begins a *For* loop that iterates once for each of the possible 40 objects that may make up a scene. Within this loop, the procedure first asks for the name of the object file and reads this from the keyboard, appends the extension *.DAT* and writes the result to the disk file. It asks for the object color and writes the return from *NumberResponse* (a number from 0 to 9) to the disk file. It asks for scale factors and writes the three, real numbers returned to the disk file, then asks for rotation factors and writes the three real numbers returned to the

disk file. It asks for translate factors and writes the three, real numbers returned to the disk file. The procedure asks if this object should be reflected in mirrored objects. It writes the Boolean result obtained from *Response* to the disk file. The procedure asks if this object may be sorted, and writes the Boolean result obtained from *Response* to the disk file. The procedure then asks if this object is a mirror and writes the Boolean result obtained from *Response* to the disk file. The procedure asks if another object should be added to the scene, and if the answer is no, the procedure leaves the loop, closes the disk file, and the entire program ends. Otherwise another iteration of the loop takes place.

{

Program for Generation of Scene Files for Model.Pas

FORMAT :

A description of the XXXXXXXX.Scn file. This is the first line.

TRUE	0	0	500	500	Do Perspective	Mx	My	Mz	D
245	25				Viewing Angle and Tilt				
45	45				Light Angle and Tilt				
FALSE					Perform Vertical Sort				
FALSE					Place Edge Reflectors				
FALSE					Place Edge Reflectors at Zero				

Grid.Dat					Object File Name
7					Object Color
50.0	50.0	50.0			Scale (Sx Sy Sz)
0.0	0.0	0.0			Rotate (Rx Ry Rz)
50.0	50.0	0.0			Translate (Tx Ty Tz)
FALSE					Object is reflected in mirrors
FALSE					Allow Sort of Object
TRUE					Object is a mirror

}

CREATING A SCENE FILE

Uses

Crt;

Const

MaxObjects = 30;

{

Keyboard Response Routines

Response - returns true or false to a Y or N keyboard stroke

NumberResponse - returns a 0 to 9 based on keyboard stroke

}

Function Response : Boolean;

Var

ch, c : Char;

Begin

Repeat

Repeat

ch := ReadKey

Until (ch<>#0);

c := UpCase(ch);

Until (c='Y') Or (c='N');

WriteLn(c);

If (c='Y') Then

Response := True

Else

Response := False

End;

Function NumberResponse : Byte;

Var

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
c : Char;
v : Byte;

Begin
  Repeat
    Repeat
      c := ReadKey;
    Until (c<>#0);
  Until (c='0') Or (c='1') Or (c='2') Or (c='3') Or (c='4')
    Or(c='5') Or (c='6') Or (c='7') Or (c='8') Or (c='9');
  v := Ord(c) - 48;
  WriteLn(v);
  NumberResponse := v
End;
```

```
{
```

Save Scene Data

```
GetSceneFile - get name of scene file
SaveScene    - saves description of a scene
}
```

```
Type
  Name = String[32];
```

```
Var
  TextDiskFile : Text;
  SceneFile    : Name;
```

```
Procedure GetSceneFile;
```

```
  Var
    x, y : Byte;
```

```
  Begin
    WriteLn;
    Write('Enter File Name => ');
    x := WhereX;
```


CREATING A SCENE FILE

```
y := WhereY;  
ReadLn(SceneFile);  
If (SceneFile = '') Then  
  Begin  
    SceneFile := 'Test';  
    GotoXY(x,y);  
    Write(SceneFile)  
  End;  
WriteLn;  
WriteLn  
End;
```

```
Procedure SaveScene(FileName : Name);
```

```
Label 1;
```

```
Var
```

```
  Descrip           : String[80];  
  NumbResp          : Integer;  
  BoolResp          : Boolean;  
  R1, R2, R3, R4    : Real;  
  I1, I2, I3, I4    : Integer;  
  ObjectsName       : Name;  
  T                 : Integer;
```

```
Begin
```

```
  FileName := FileName + '.Scn';  
  Assign(TextDiskFile, FileName);  
  Rewrite(TextDiskFile);
```

```
  Write('Description of the Scene => ');  
  ReadLn(Descrip);  
  WriteLn(TextDiskFile, Descrip);
```

```
  WriteLn(TextDiskFile);
```

```
  Write('Perspective Display? ');  
  BoolResp := Response;  
  Write(TextDiskFile, BoolResp, '');
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
If BoolResp Then
    Begin
        Write('(Mx My Mz D) = ');
        ReadLn(I1,I2,I3,I4);
        WriteLn(TextDiskFile,I1:3,' ',I2:3,' ',I3:3,'
',I4:3)
    End
Else
    WriteLn(TextDiskFile,'0.0 0.0 0.0 0.0');

WriteLn('Direction for Viewing');
Write(' Around the z-Axis = ');
ReadLn(NumbResp);
Write(TextDiskFile,NumbResp,' ');
Write(' Off of the z-Axis = ');
ReadLn(NumbResp);
WriteLn(TextDiskFile,NumbResp);

WriteLn('Direction of Light');
Write(' Around the z-Axis = ');
ReadLn(NumbResp);
Write(TextDiskFile,NumbResp,' ');
Write(' Off of the z-Axis = ');
ReadLn(NumbResp);
WriteLn(TextDiskFile,NumbResp);

Write('Vertical Sort of Objects? ');
WriteLn(TextDiskFile,Response);

Write('Place Edge Reflectors at (x=-100,y=-100 and
        z=-100)? ');
WriteLn(TextDiskFile,Response);

Write('Place Edge Reflectors at (x=-100,y=-100 and
        z=0)? ');
WriteLn(TextDiskFile,Response);

WriteLn;
```


CREATING A SCENE FILE

```
WriteLn('Now add objects to the scene ->');
WriteLn;

For T := 1 To MaxObjects Do
  Begin
    WriteLn(TextDiskFile);

    Write('Enter the Objects Name => ');
    ReadLn(ObjectsName);
    ObjectsName := ObjectsName + '.Dat';
    WriteLn(TextDiskFile, ObjectsName);

    Write('Color Of the Object? ');
    WriteLn(TextDiskFile, NumberResponse);

    Write('Scale Factors (Sx Sy Sz) => ');
    ReadLn(R1, R2, R3);
    WriteLn(TextDiskFile, R1:4:4, ' ', R2:4:4, ' ', R3:4:4);

    Write('Rotate Factors (Rx Ry Rz) => ');
    ReadLn(R1, R2, R3);
    WriteLn(TextDiskFile, R1:4:4, ' ', R2:4:4, ' ', R3:4:4);

    Write('Translate Factors (Tx Ty Tz) => ');
    ReadLn(R1, R2, R3);
    WriteLn(TextDiskFile, R1:4:4, ' ', R2:4:4, ' ', R3:4:4);

    Write('Reflect this Object in Mirrored Objects? ');
    WriteLn(TextDiskFile, Response);

    Write('Allow this Object to be Sorted for
          Placement?');
    WriteLn(TextDiskFile, Response);

    Write('This Object is a Mirror? ');
    WriteLn(TextDiskFile, Response);

    Write('Add another Object to Scene? ');
    If Not(Response) Then Goto 1;
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
        WriteLn;
    End;

1: Close(TextDiskFile)
    End;

Begin
    ClrScr;
    WriteLn('Program to Generate a Scene File for Model');
    WriteLn;
    GetSceneFile;
    SaveScene(SceneFile)
End.
```

Figure 10-2. Listing of the *ScnMaker.Pas* Program

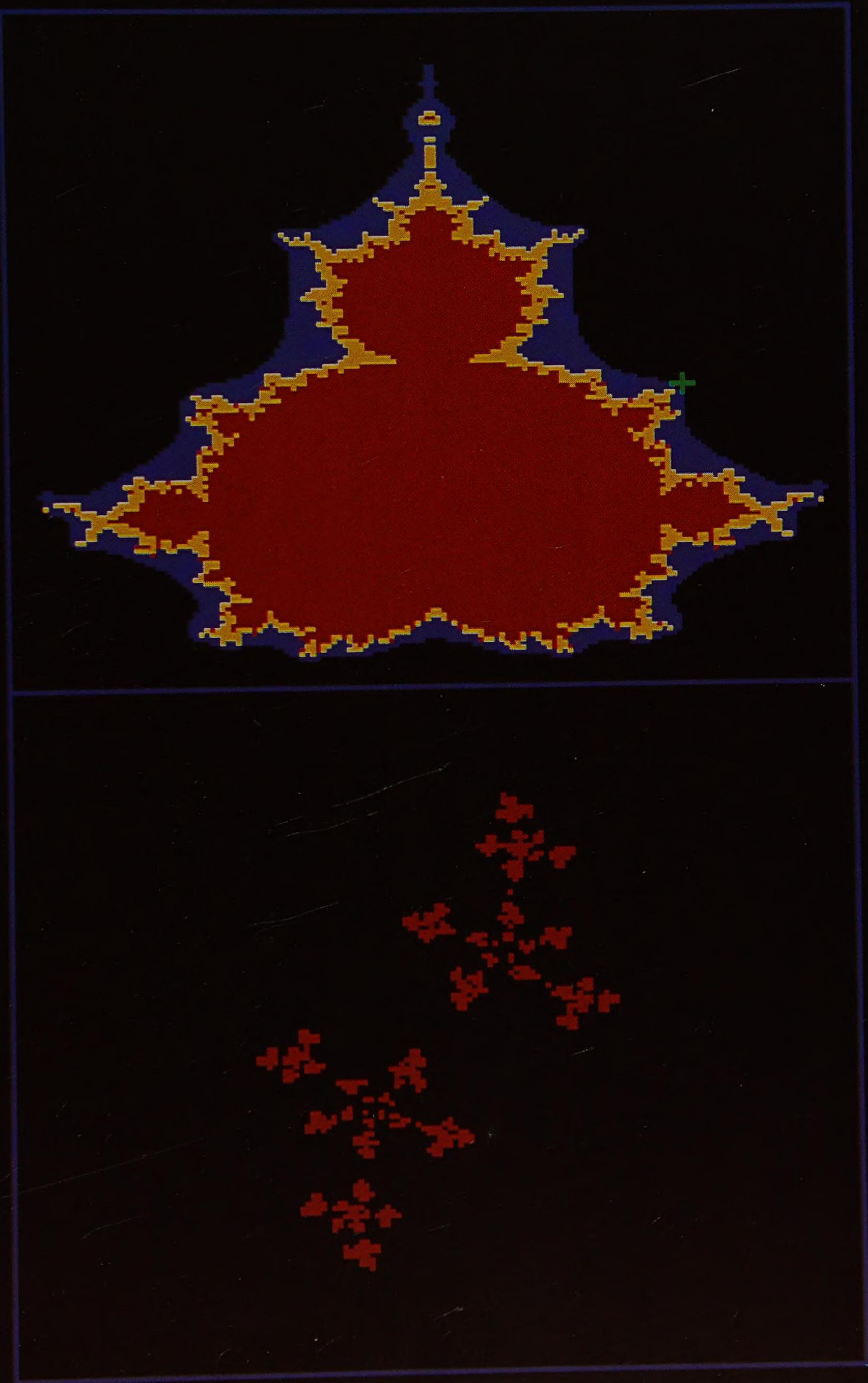


Plate 1. Walking About the Mandelbrot Set

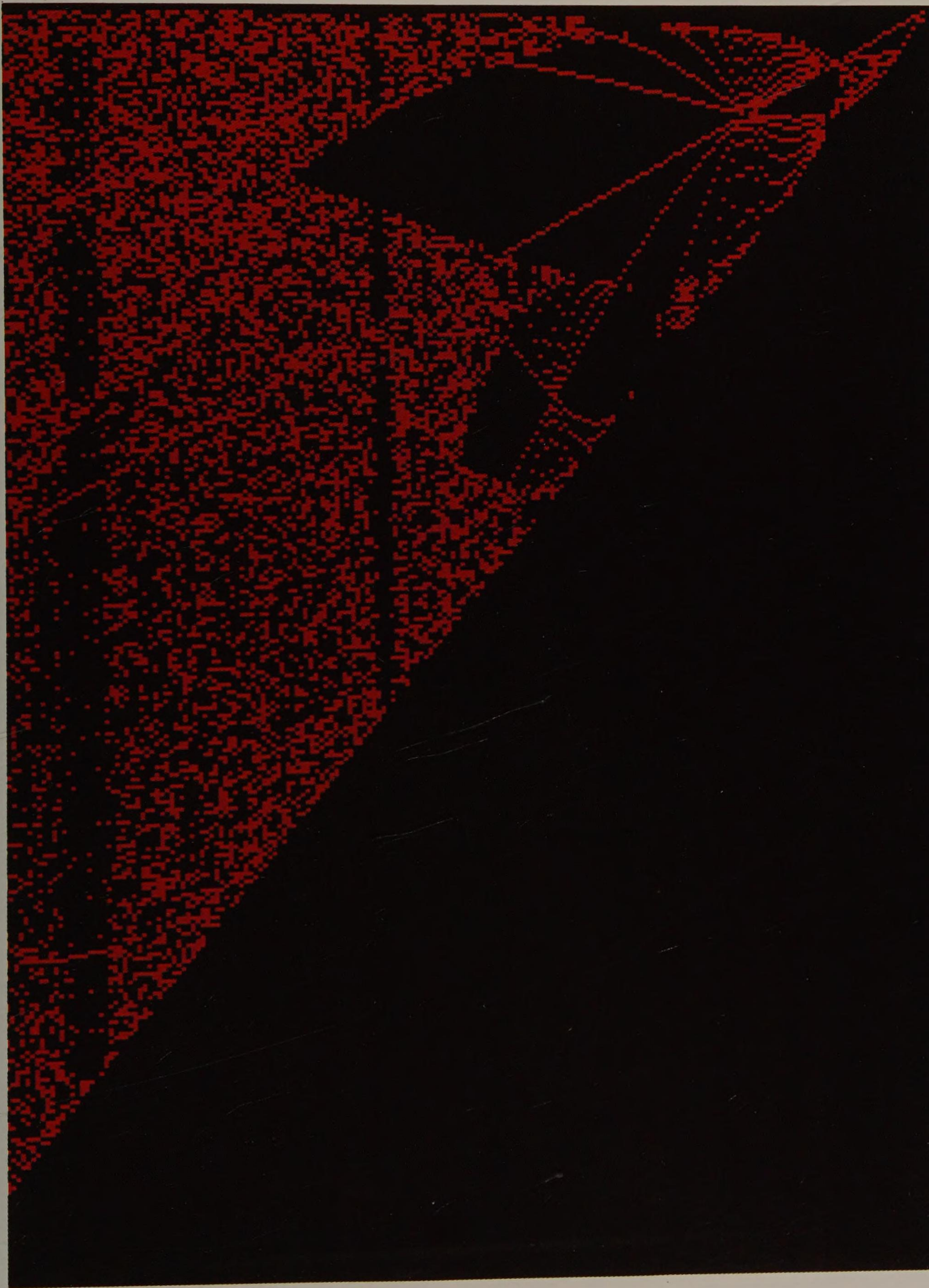


Plate 3. Bifurcation Diagram



Plate 4. A Strange Attractor in Three Dimensions

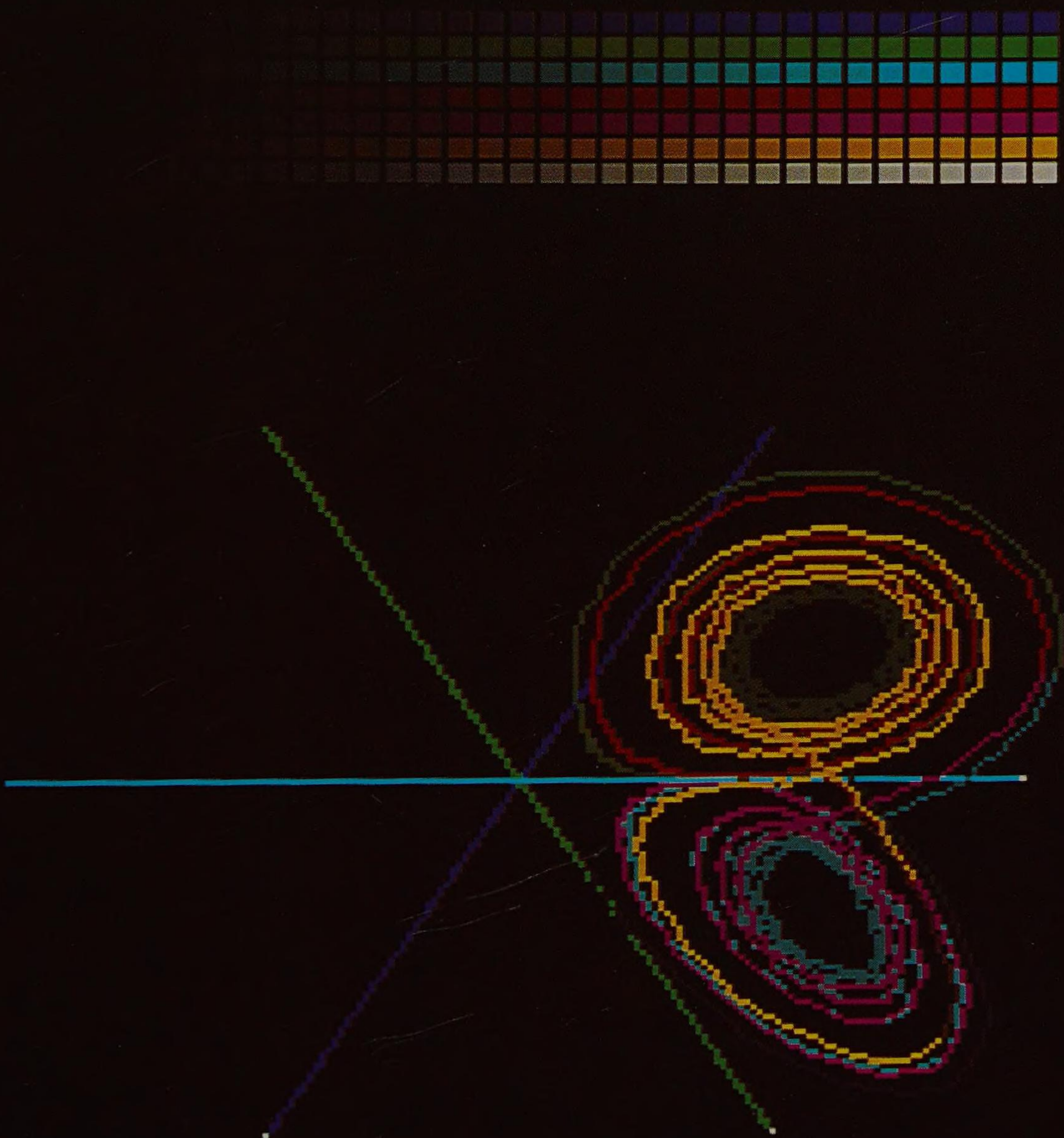


Plate 5. The Lorenz Attractor

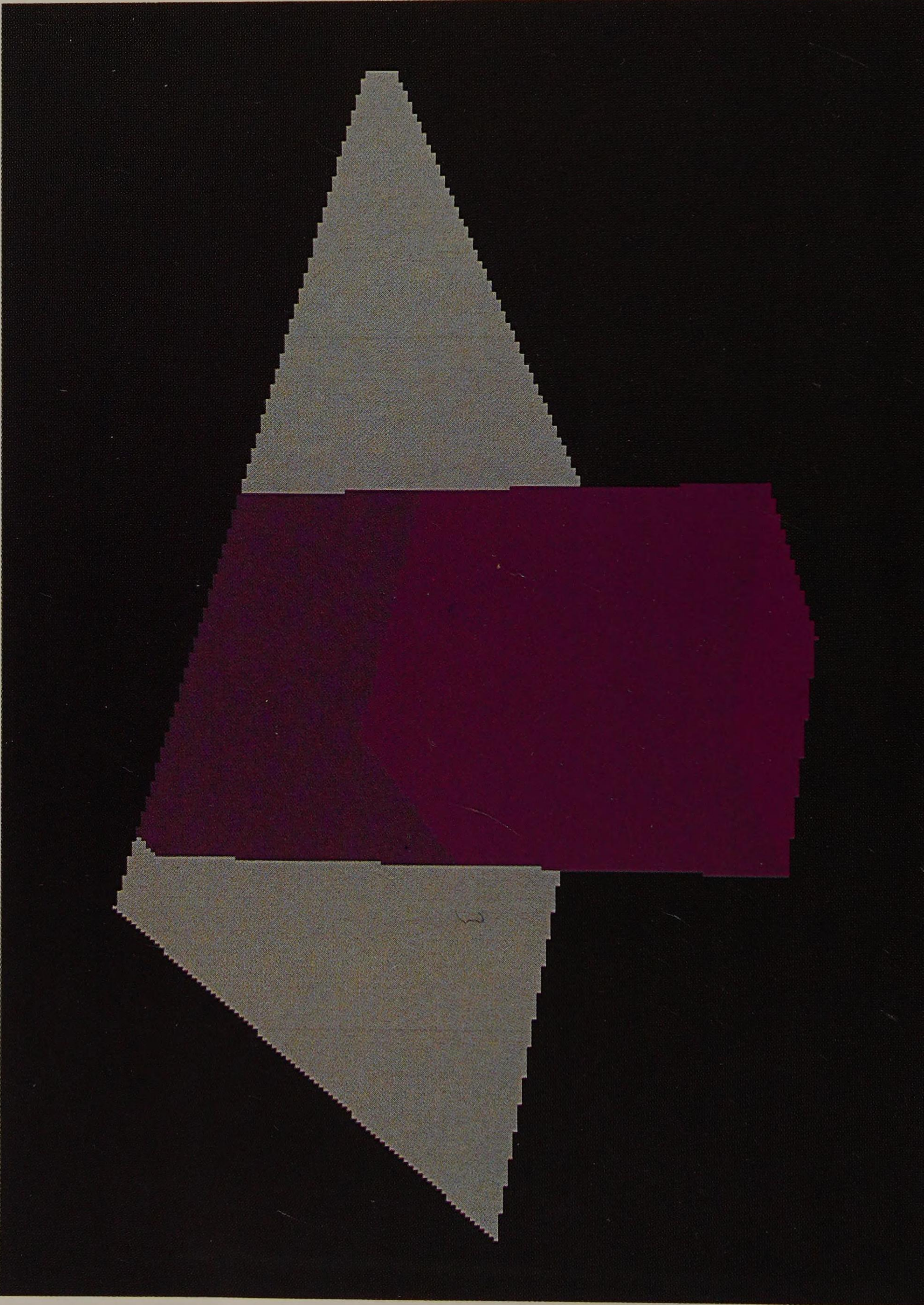


Plate 6. Solid Modeled Cube

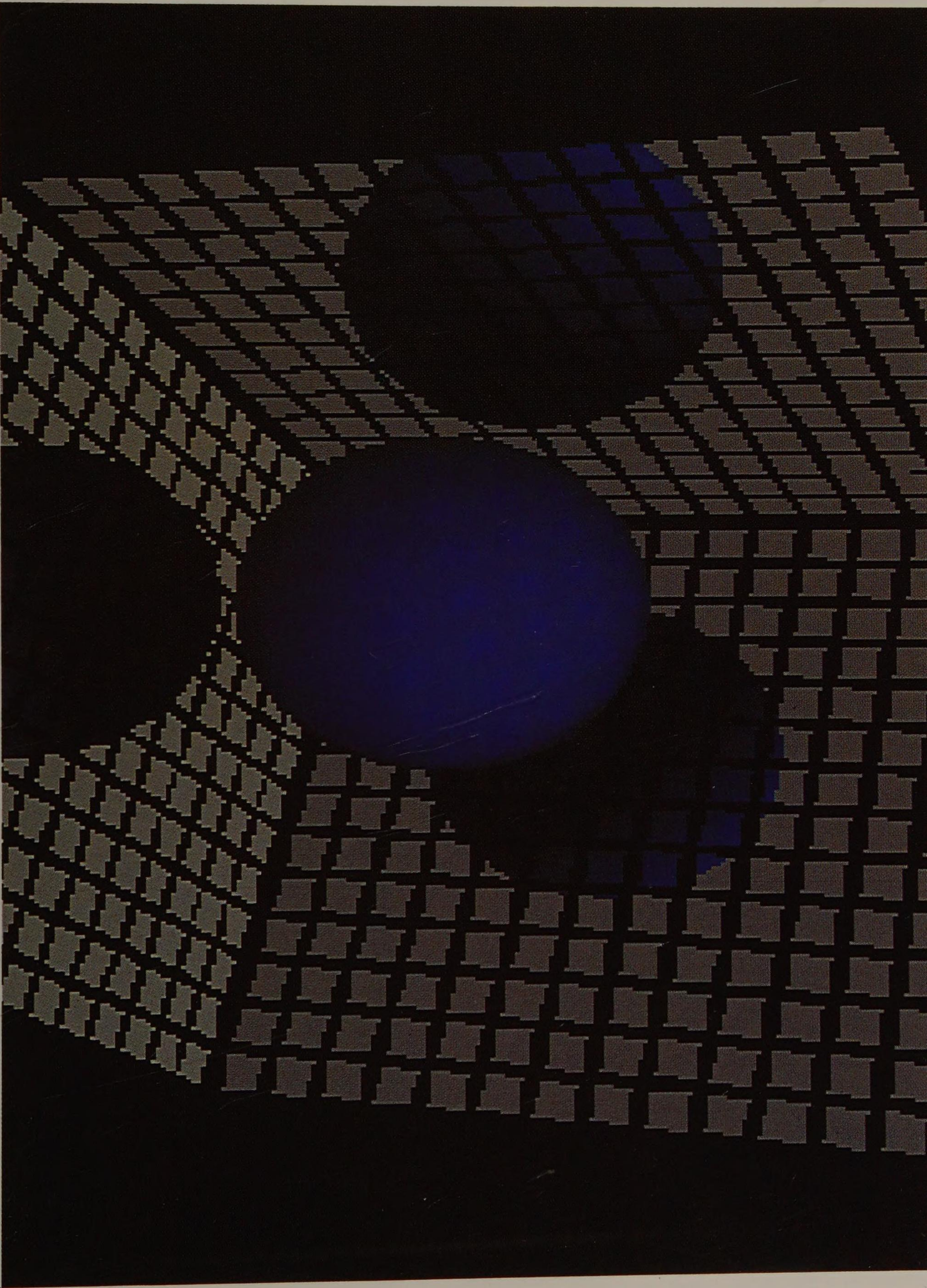


Plate 7. Solid Modeled Sphere with Mirrored Walls

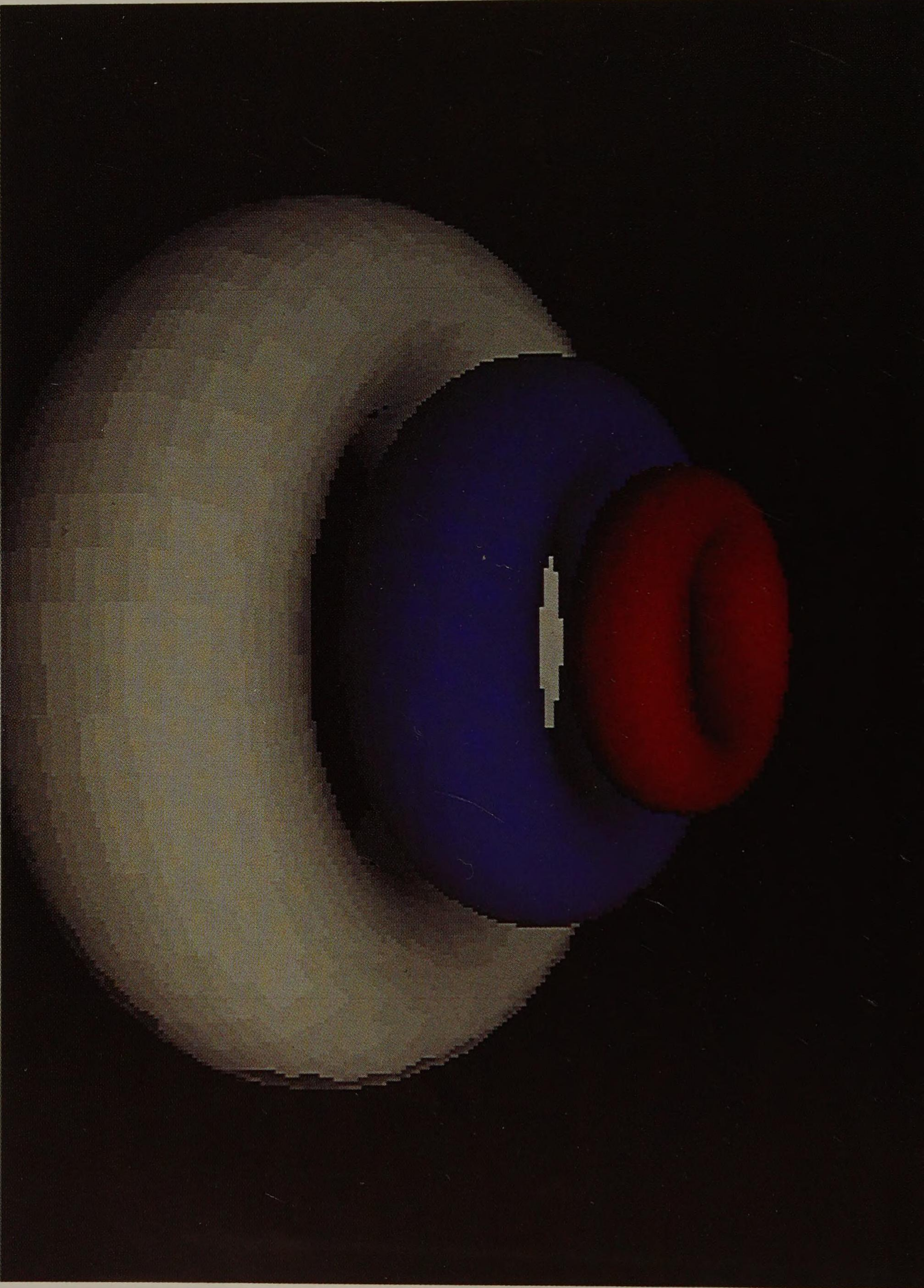


Plate 8. Solid Modeled Stacked Toroids

Plate 9. Four Columns with Spheres Using Solid Modeling

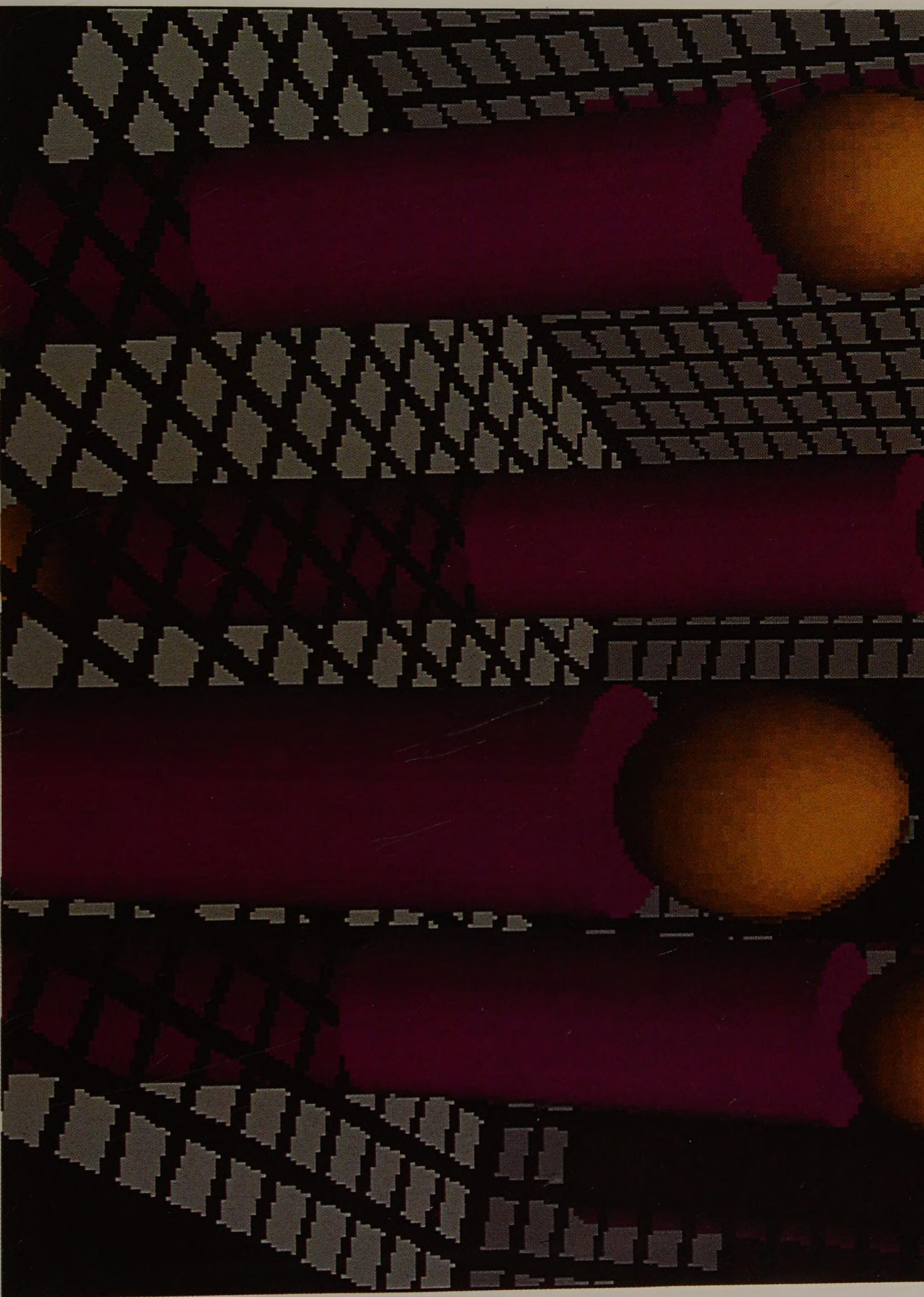


Plate 10. Well Scene Using Solid Modeling

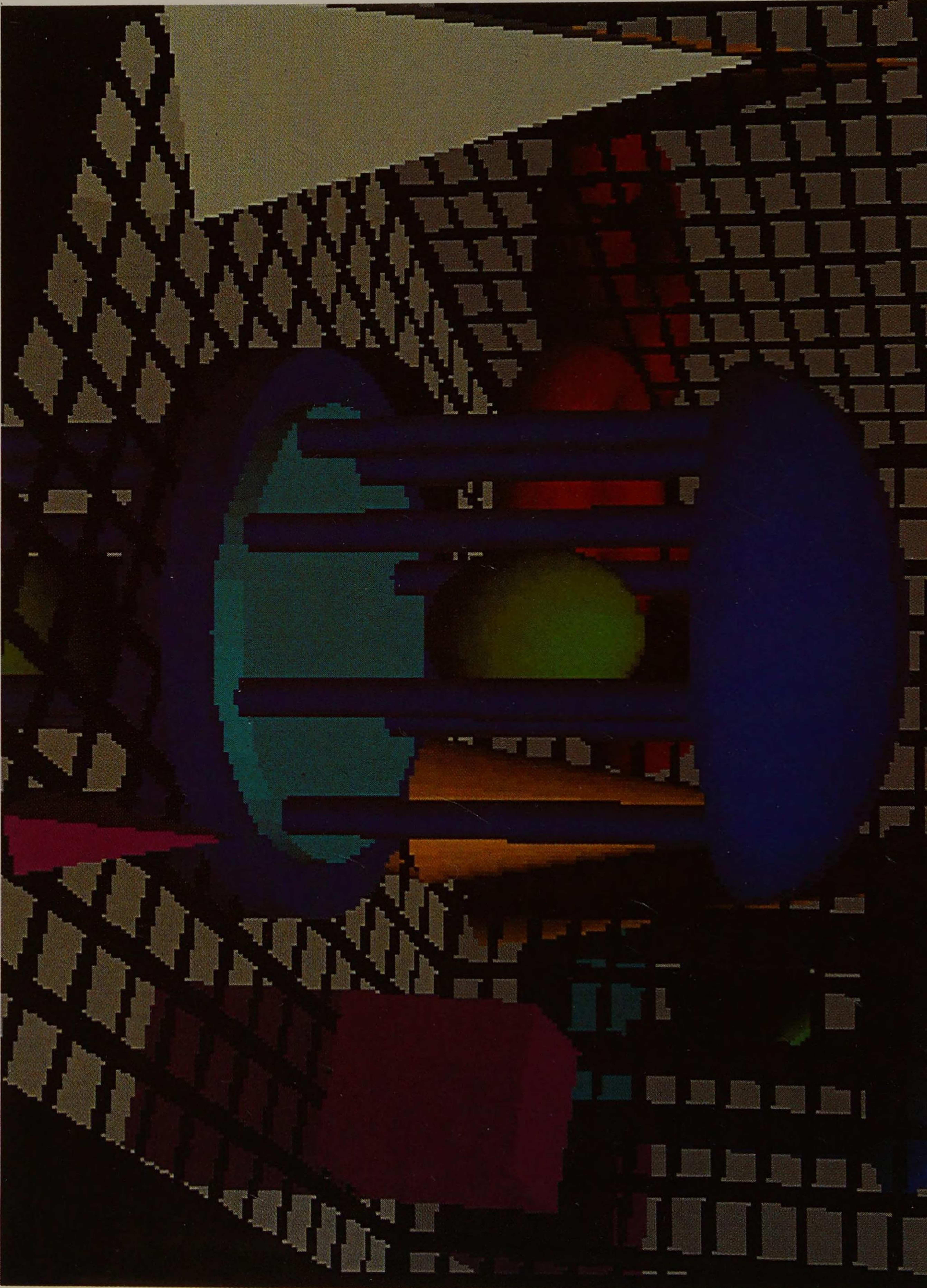
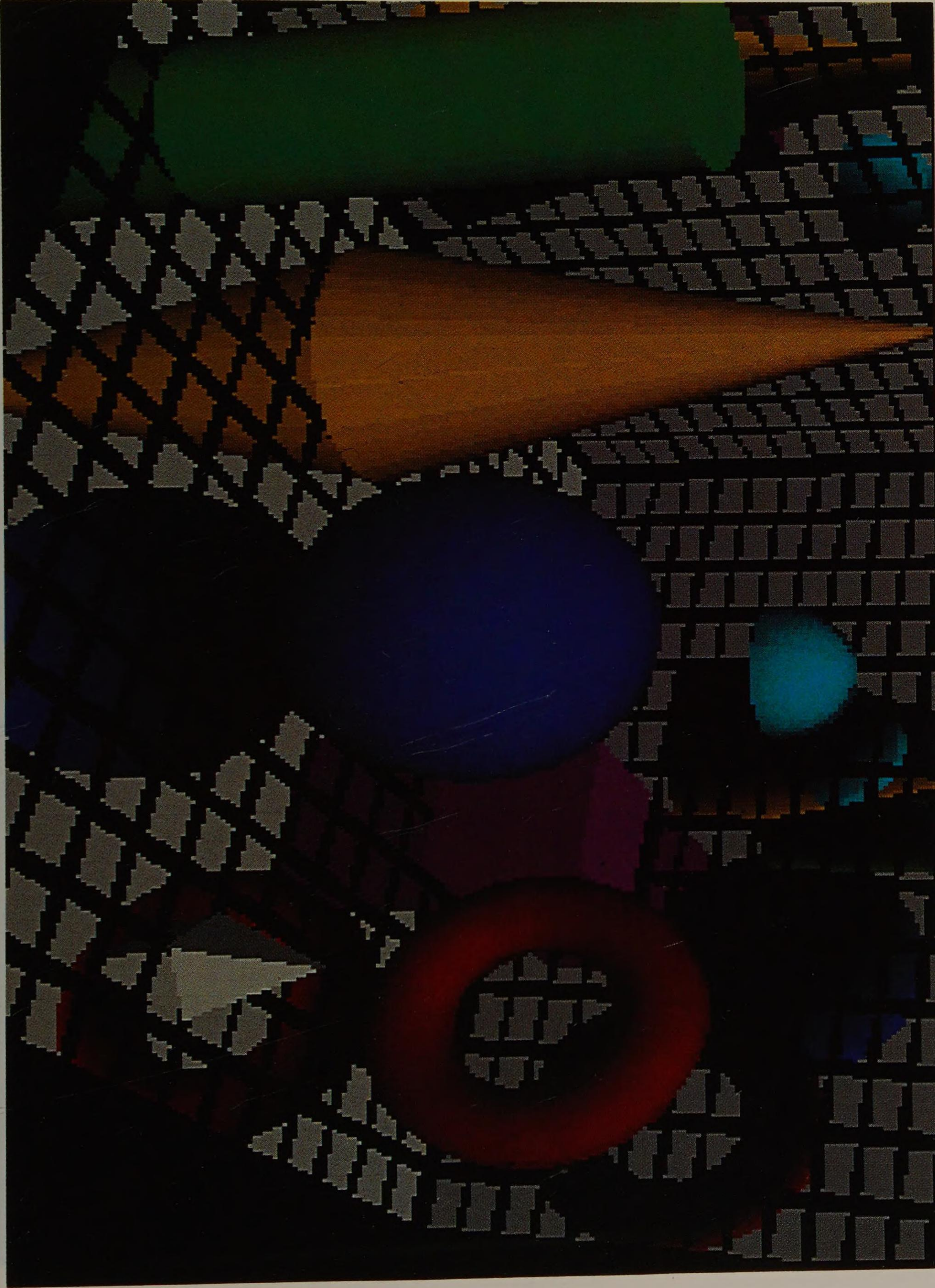


Plate 11. Collection of Solid Shapes Produced by Solid Modeling



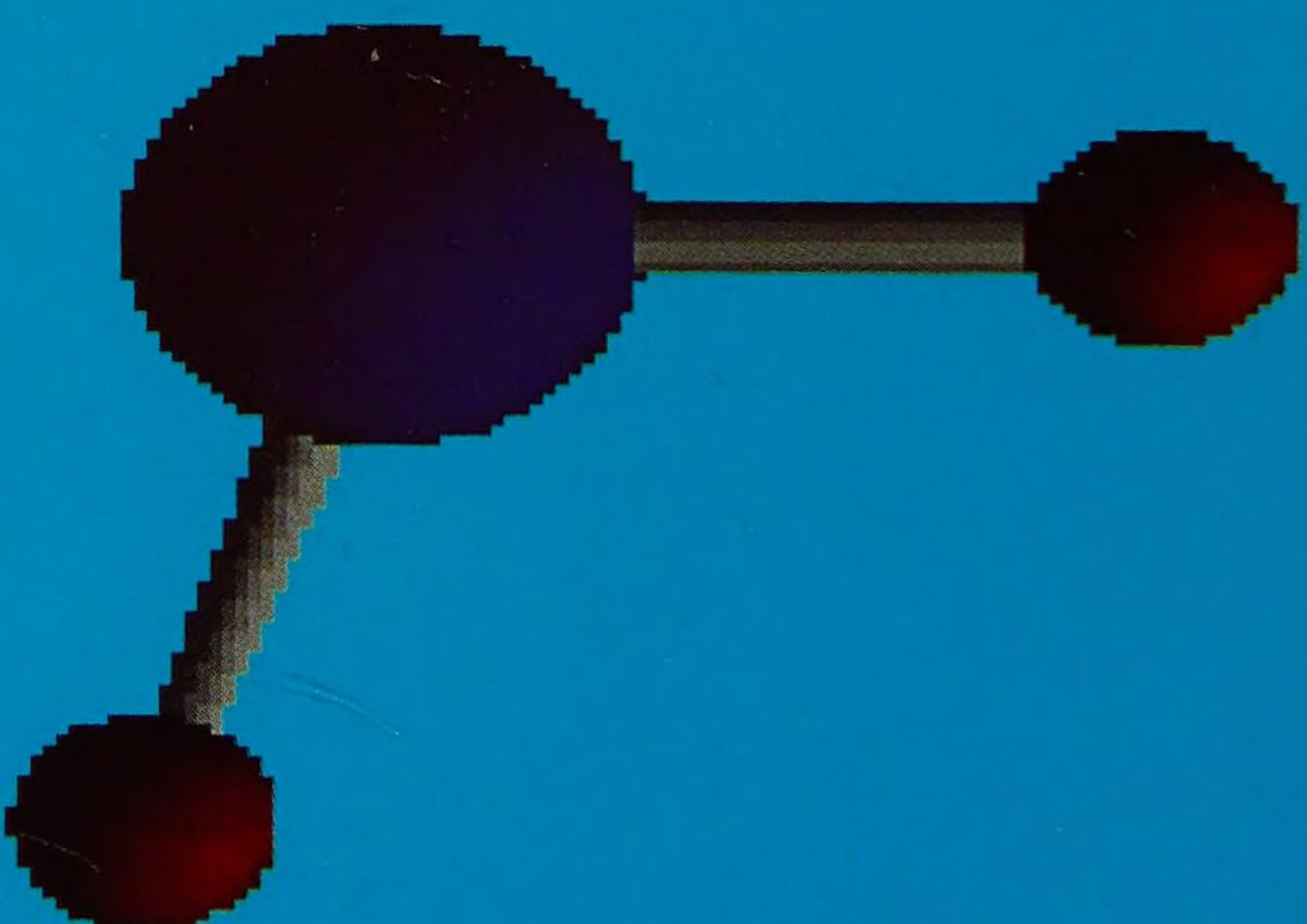


Plate 12. Model of Water Molecule Produced by Solid Modeling

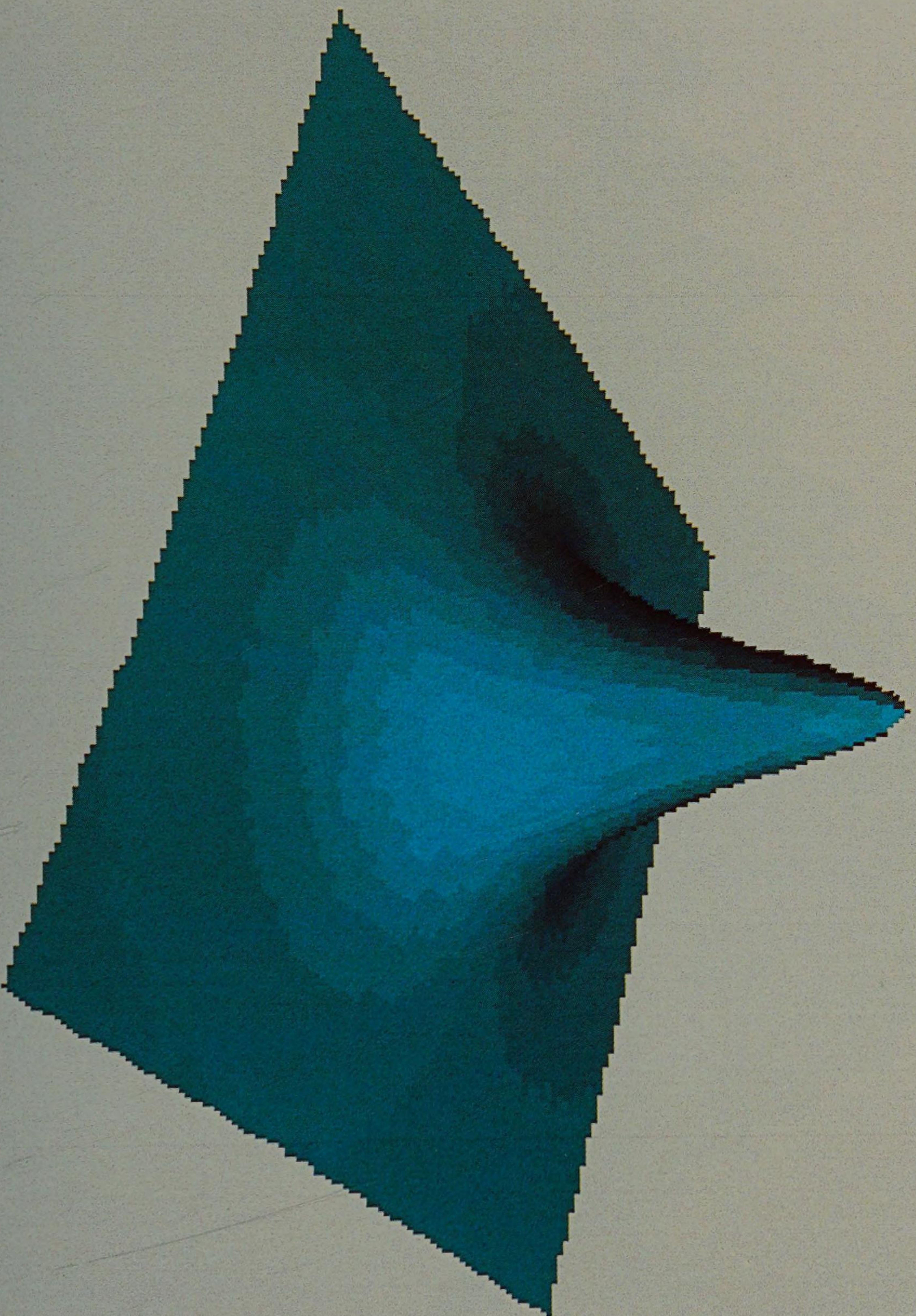


Plate 13. Plot of $z = 75 / (x^2 + y^2 + 1)$ Using Solid Modeling

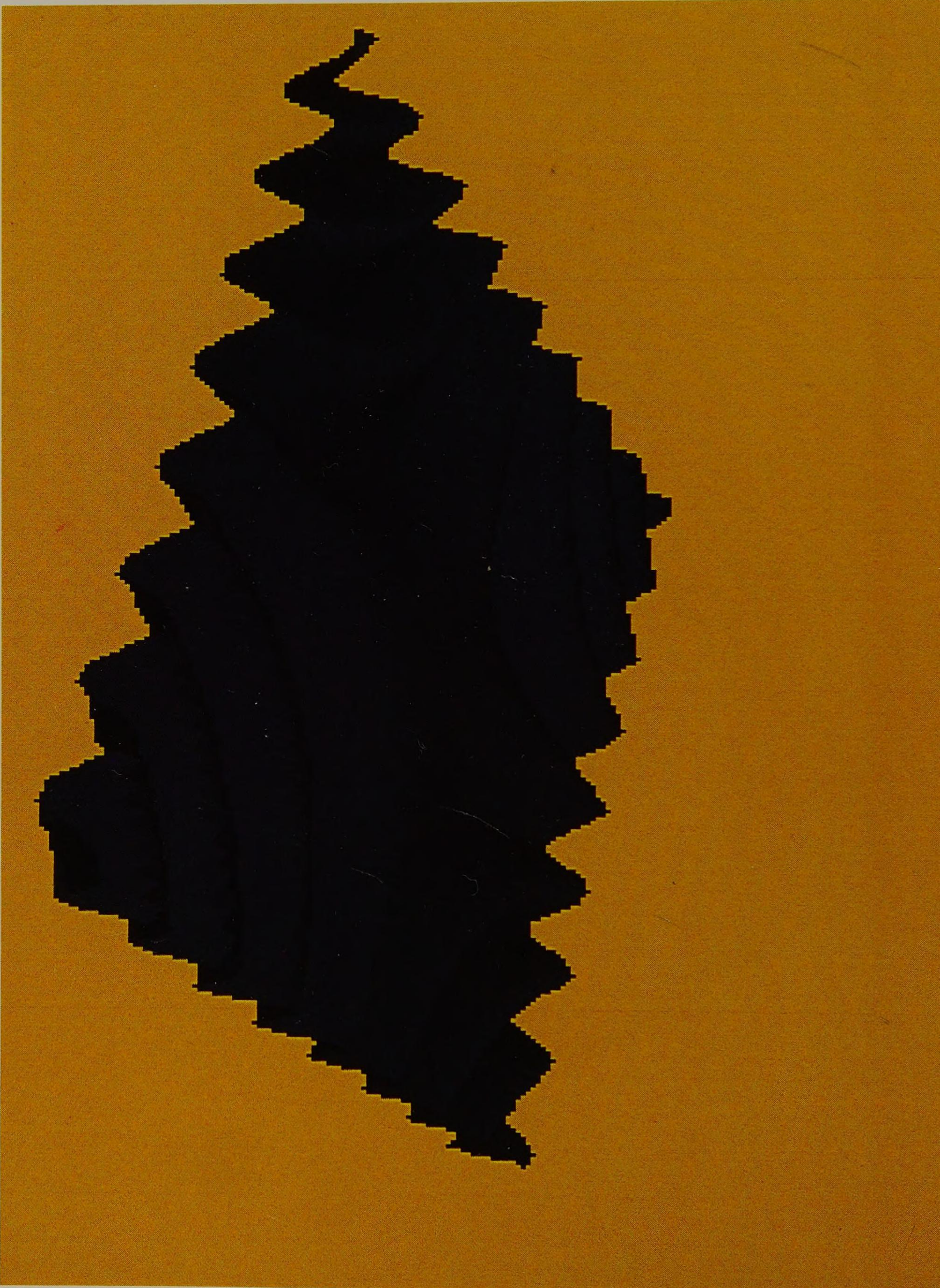


Plate 14. Plot of $z = 6 (\cos(xy) + 1)$ Using Solid Modeling

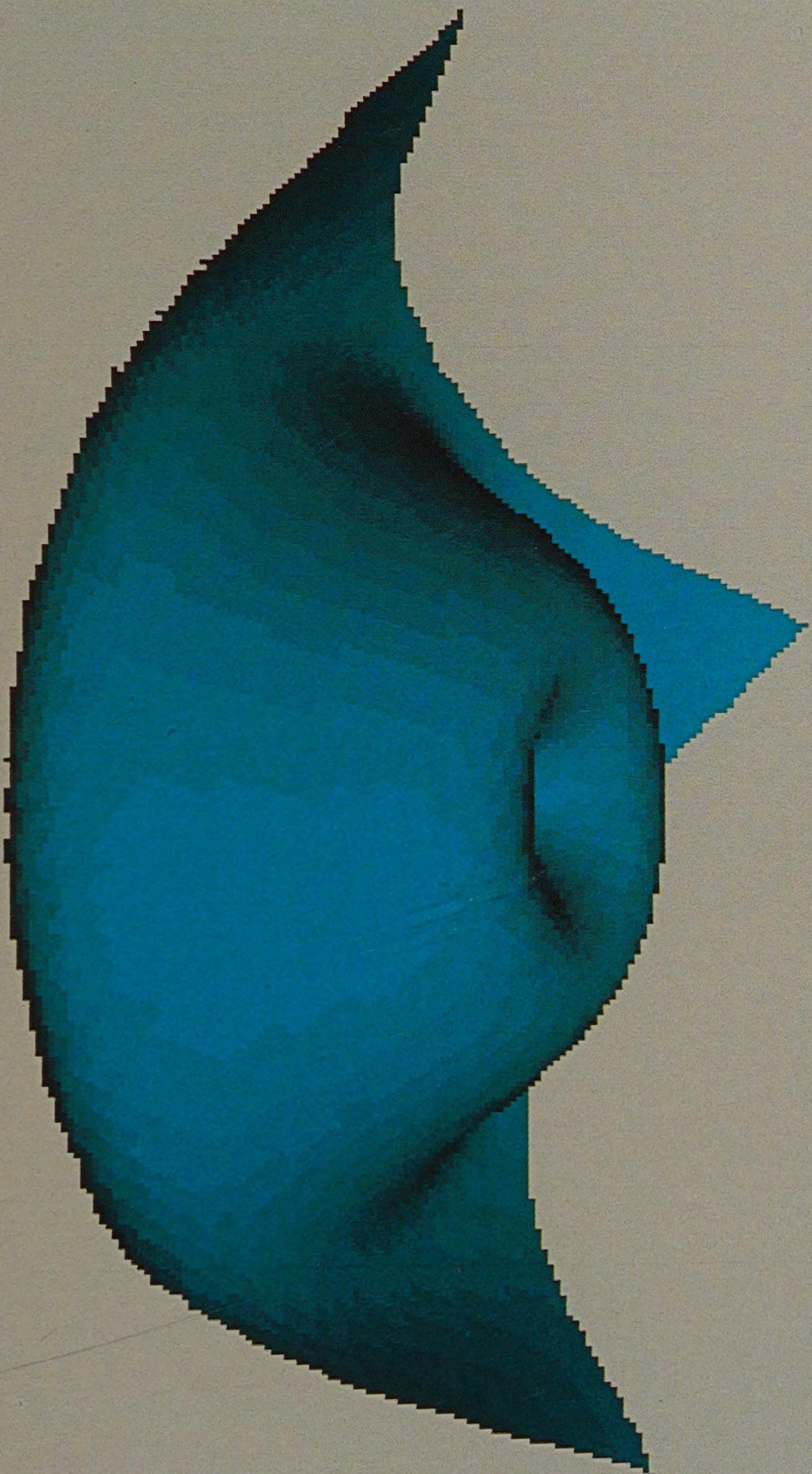


Plate 15. Plot of $z = 20(\sin(\sqrt{x^2 + y^2}) + 1)$

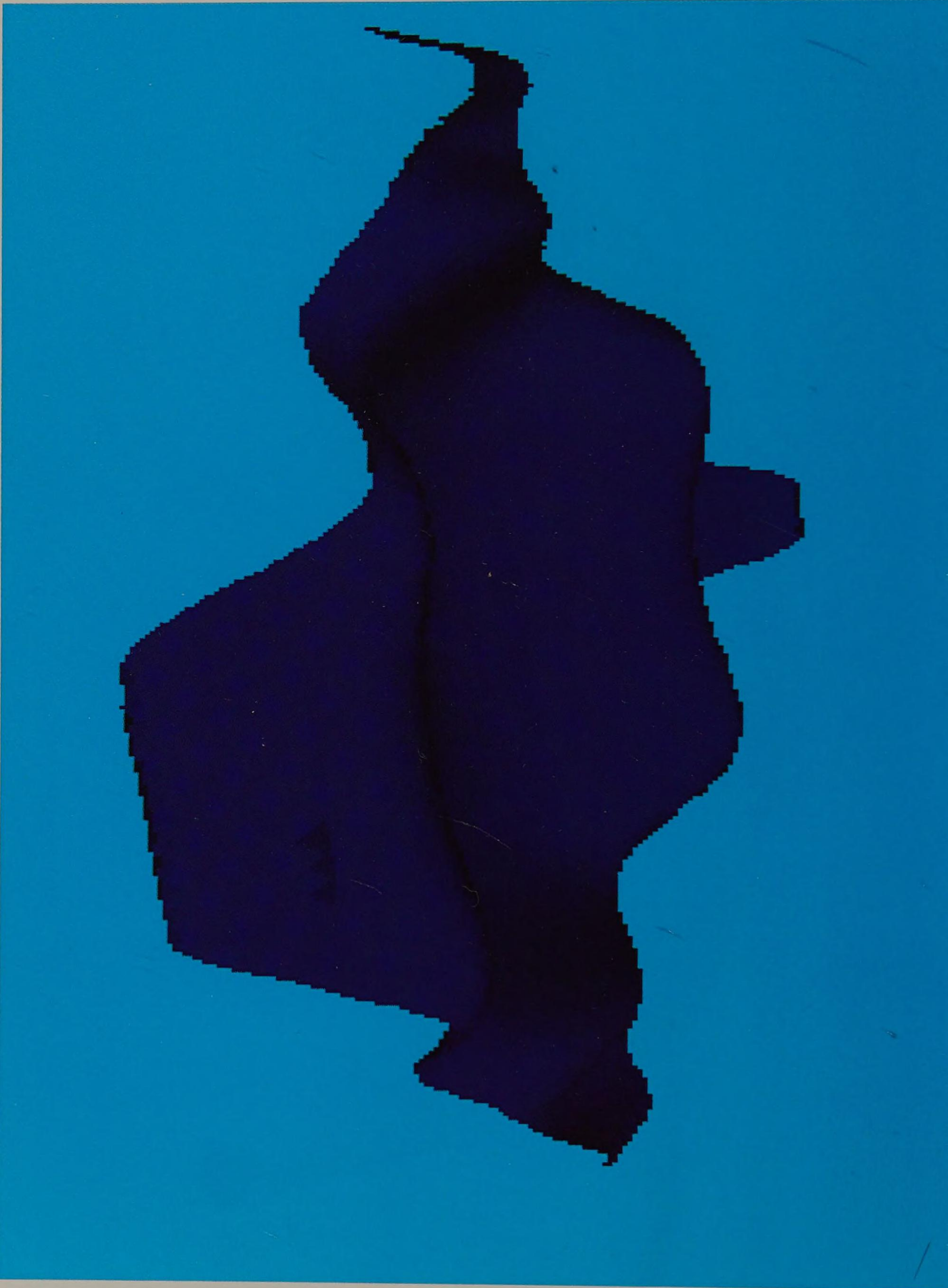


Plate 16. Plot of $z = 10(1 + \sqrt{x^2 + y^2} + \sin(0.1(x^2 + y^2)) + \sin(xy))$



Plate 17. Solid Modeling of Solids of Revolution

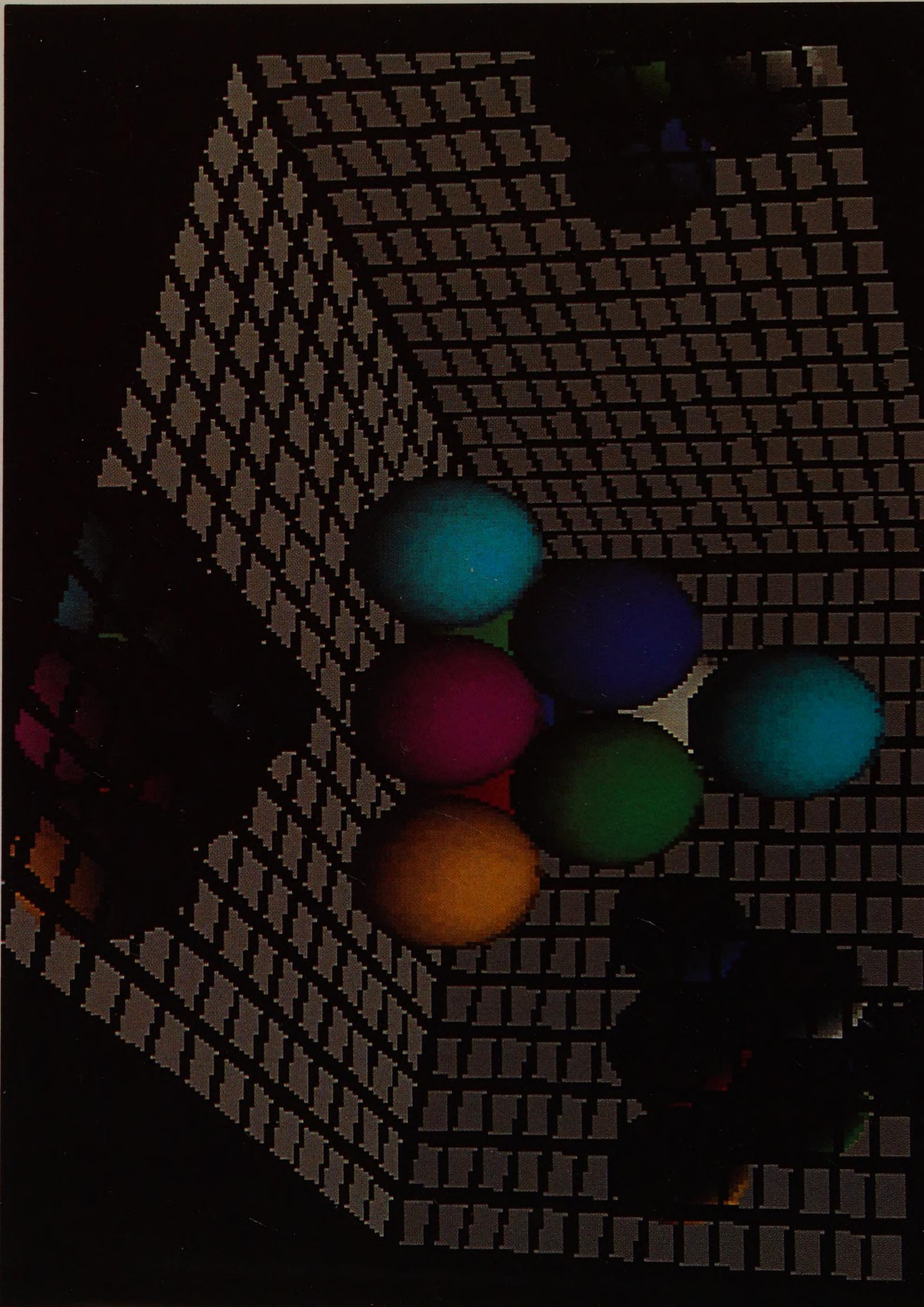


Plate 18. Solid Modeling of Stacked Spheres

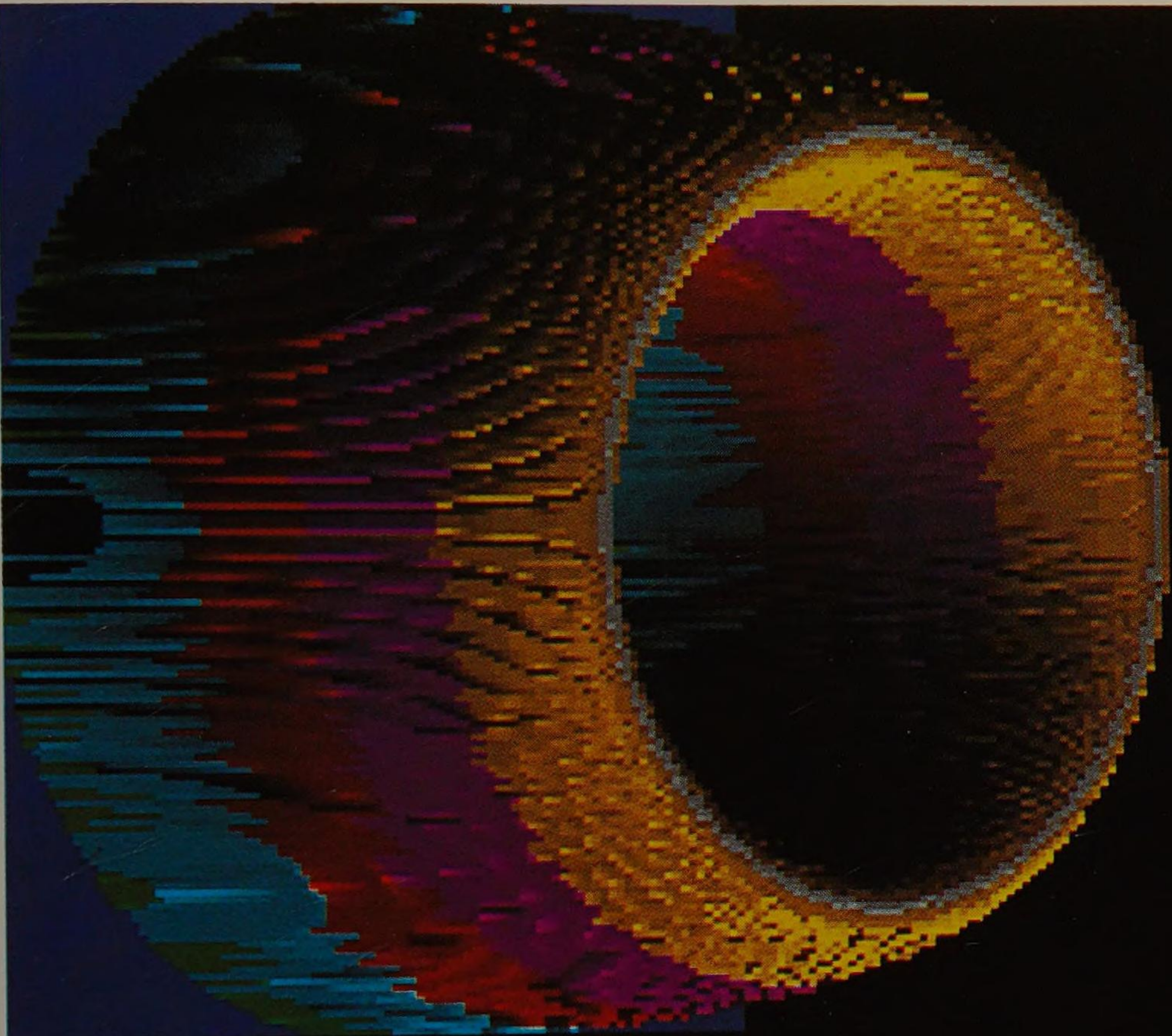


Plate 19. Bowl Rendered by Z-Buffering

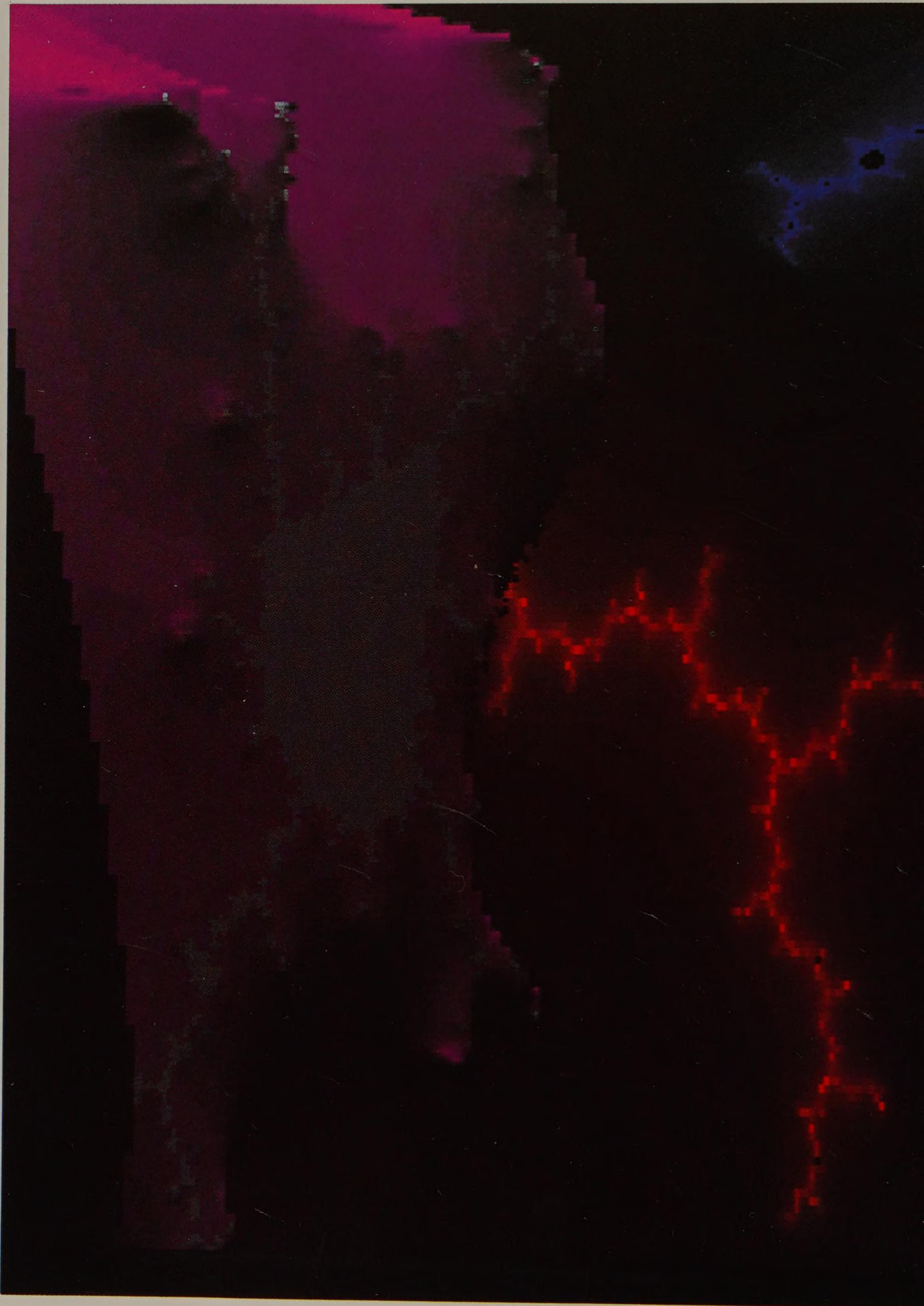


Plate 20. Three-Dimensional Mandelbrot Set



Plate 21. Three-Dimensional Julia Set

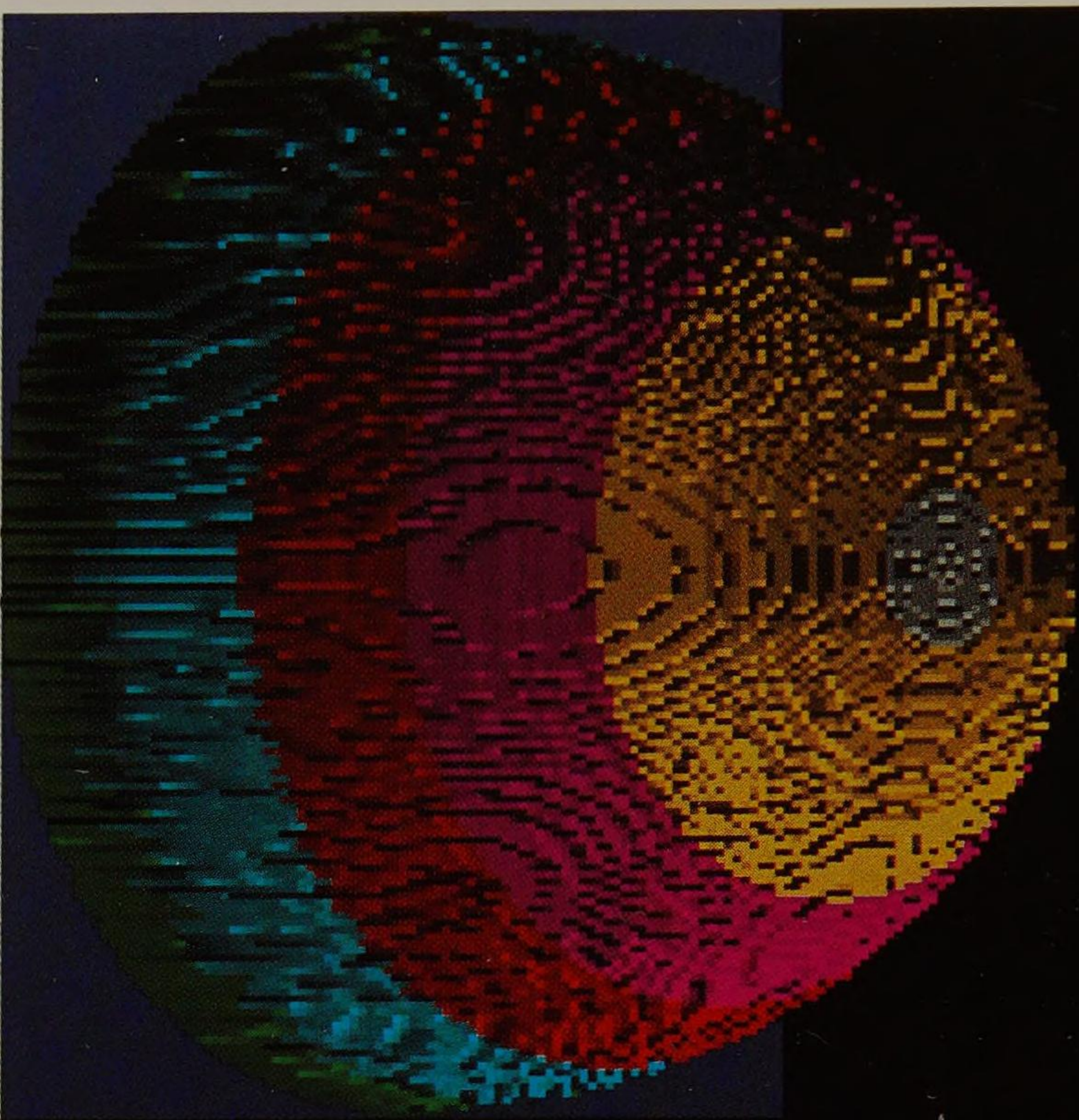


Plate 22. Hemisphere Rendered by Z-Buffering

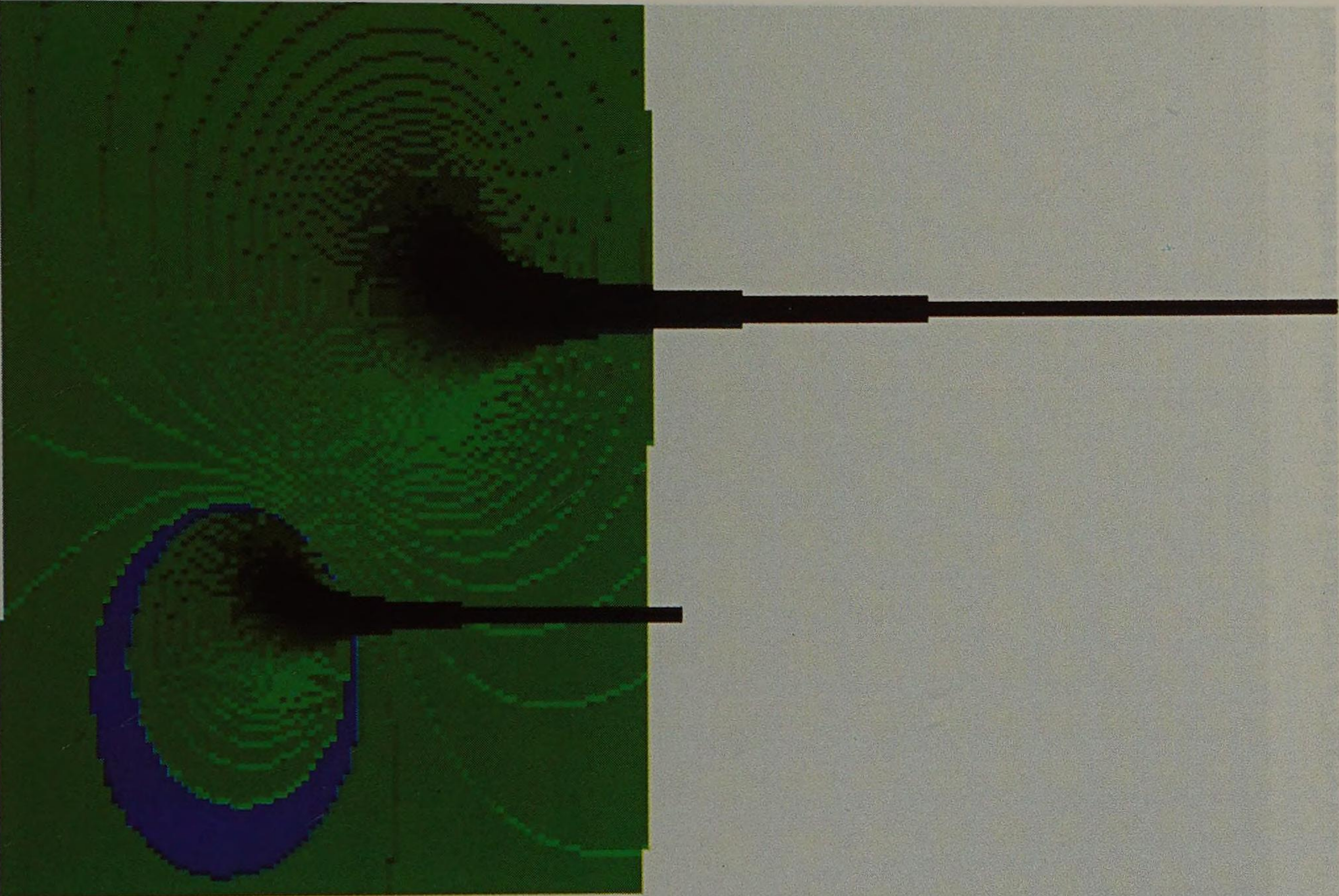


Plate 23. Magnetic Field Between Two Parallel Wires

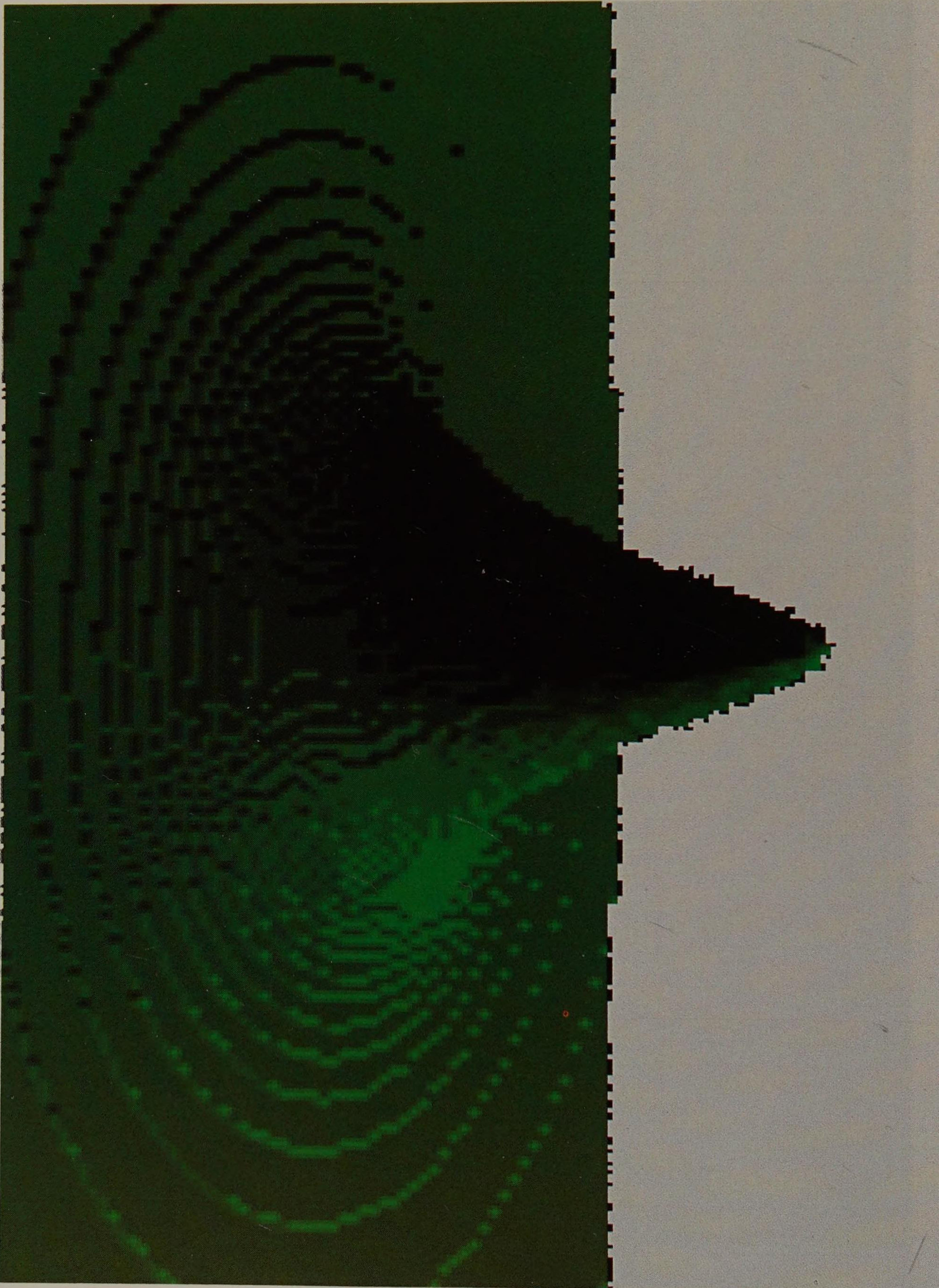
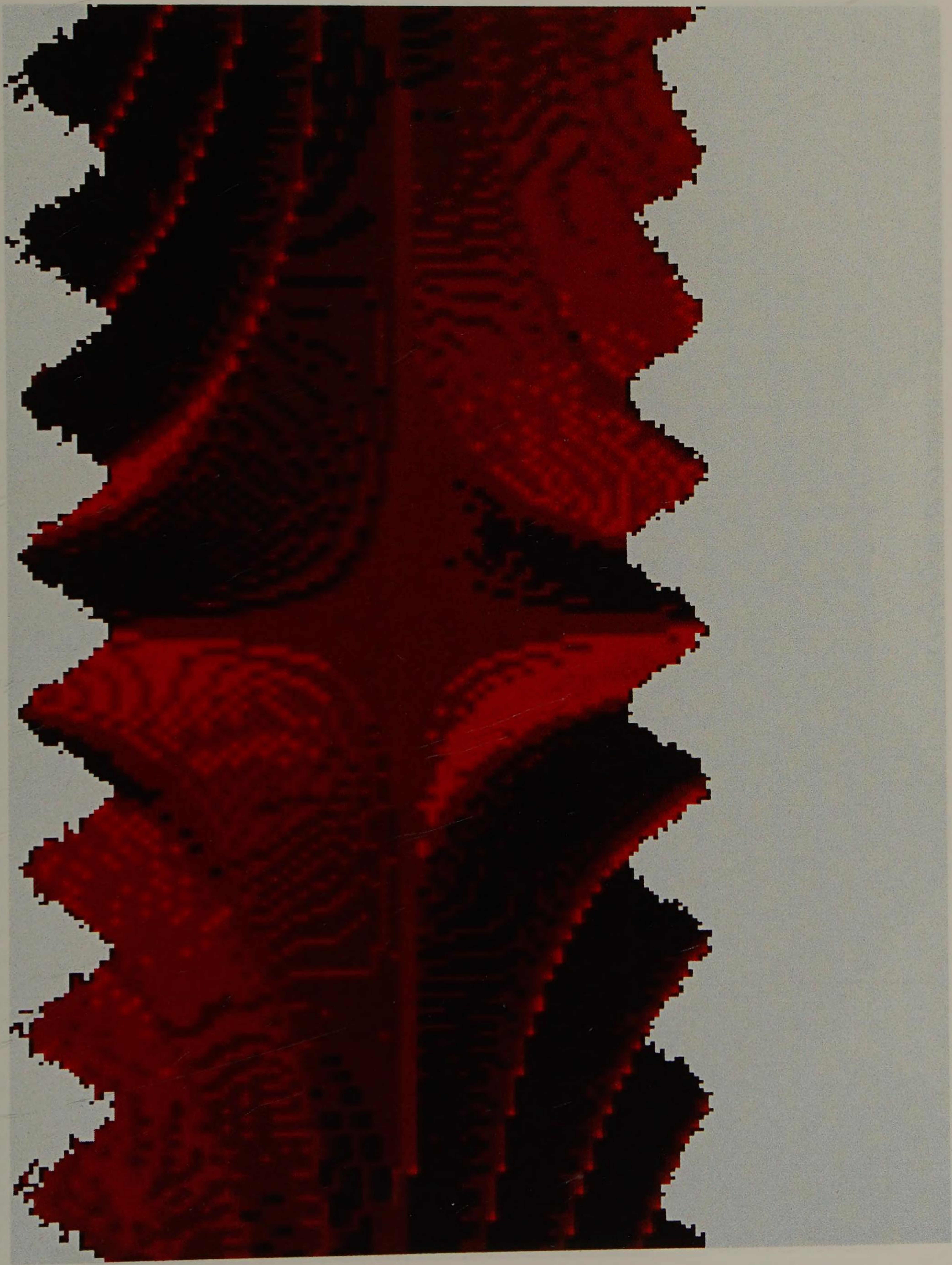


Plate 24. Z-Buffered Rendering of the Equation $z = 75(x^2 + y^2 + 1)$

Plate 25. Z-Buffered Rendering of the Equation $z = 6(\cos(xy) + 1)$



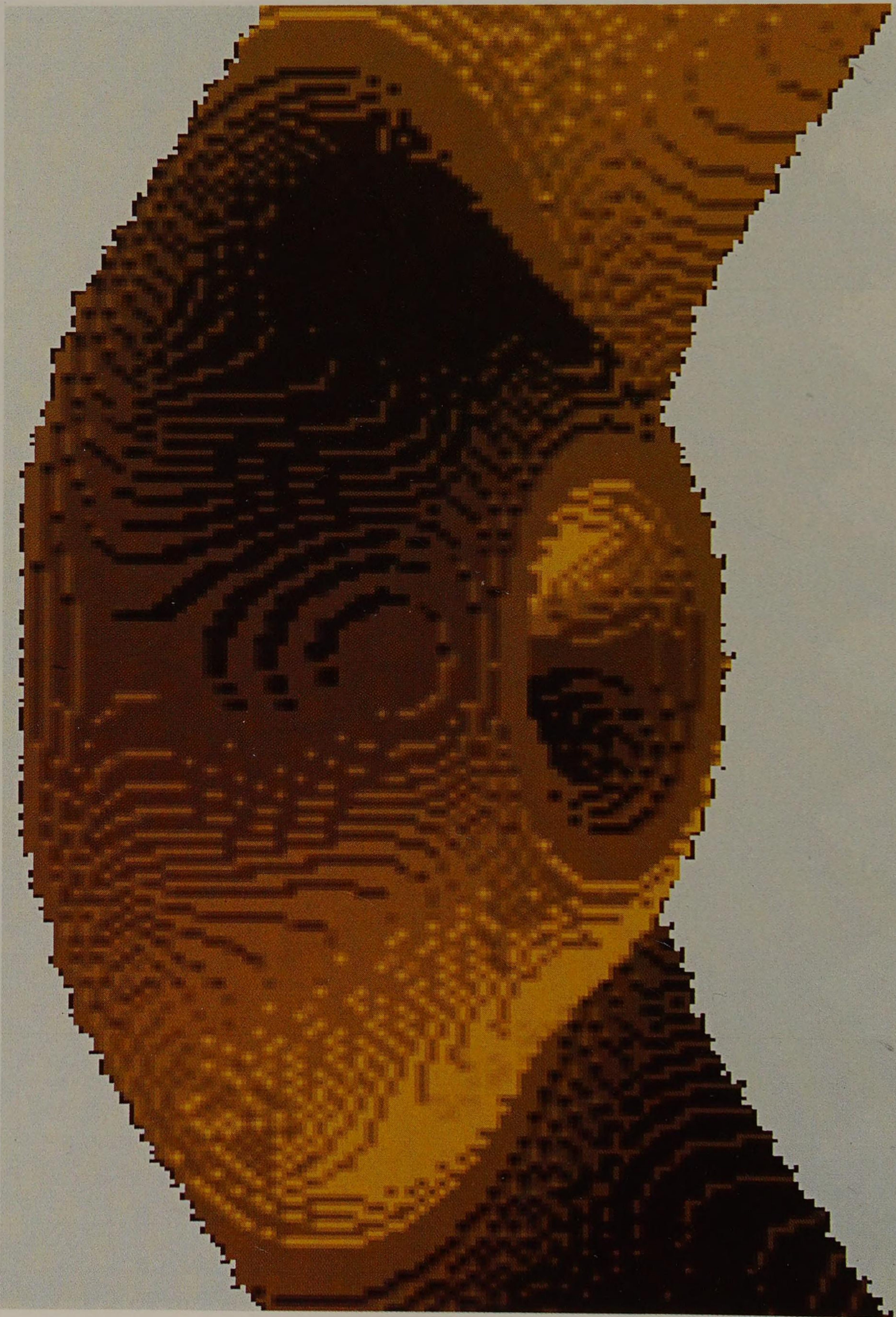


Plate 26. Z-Buffered Rendering of the Equation $z = 20(\sin(\sqrt{x^2 + y^2}) + 1)$

Plate 27. Z-Buffered Rendering of the Equation $z = 10(1 + \sqrt{x^2 + y^2} + \sin(0.1(x^2 + y^2)) + \sin(xy))$



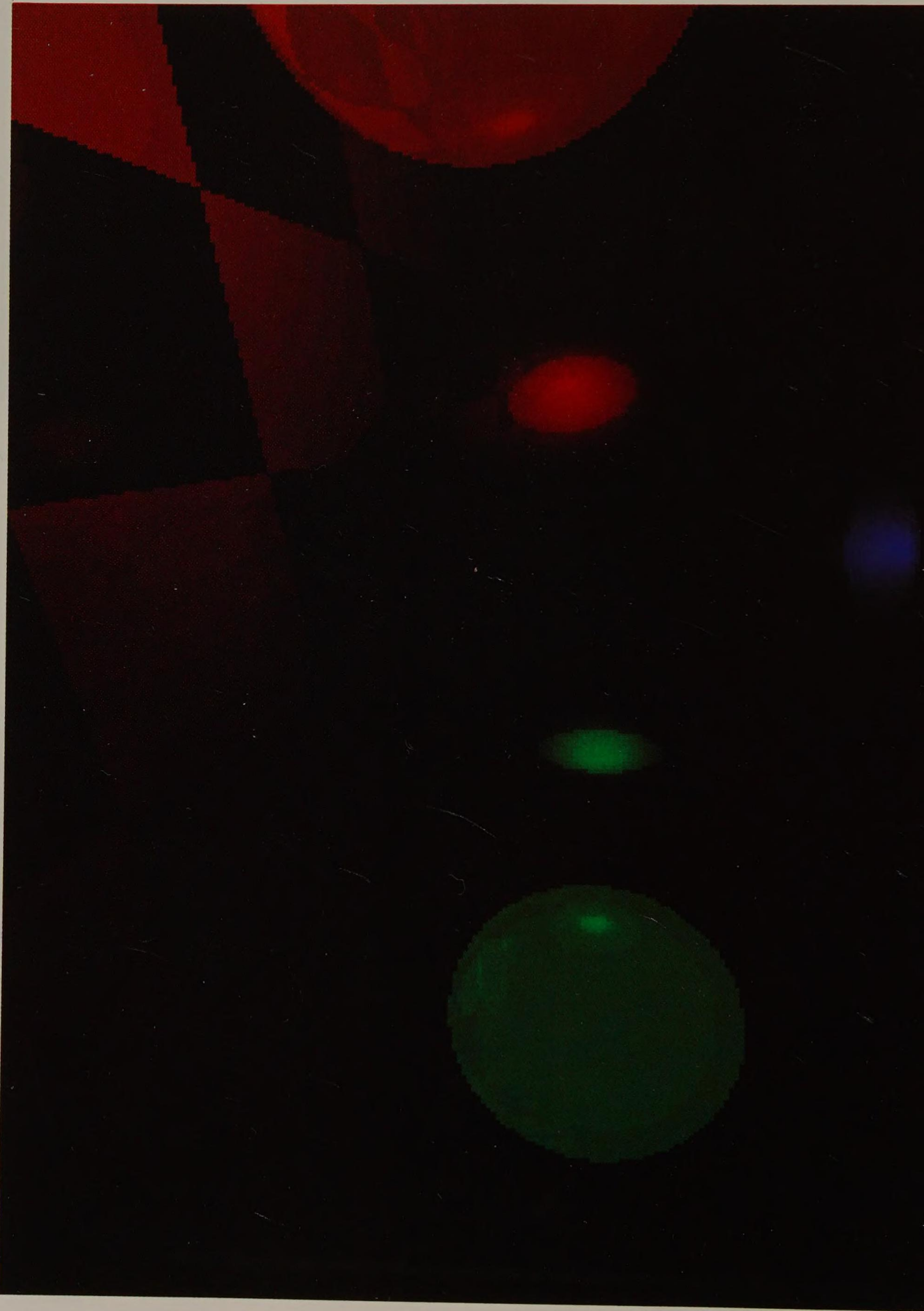


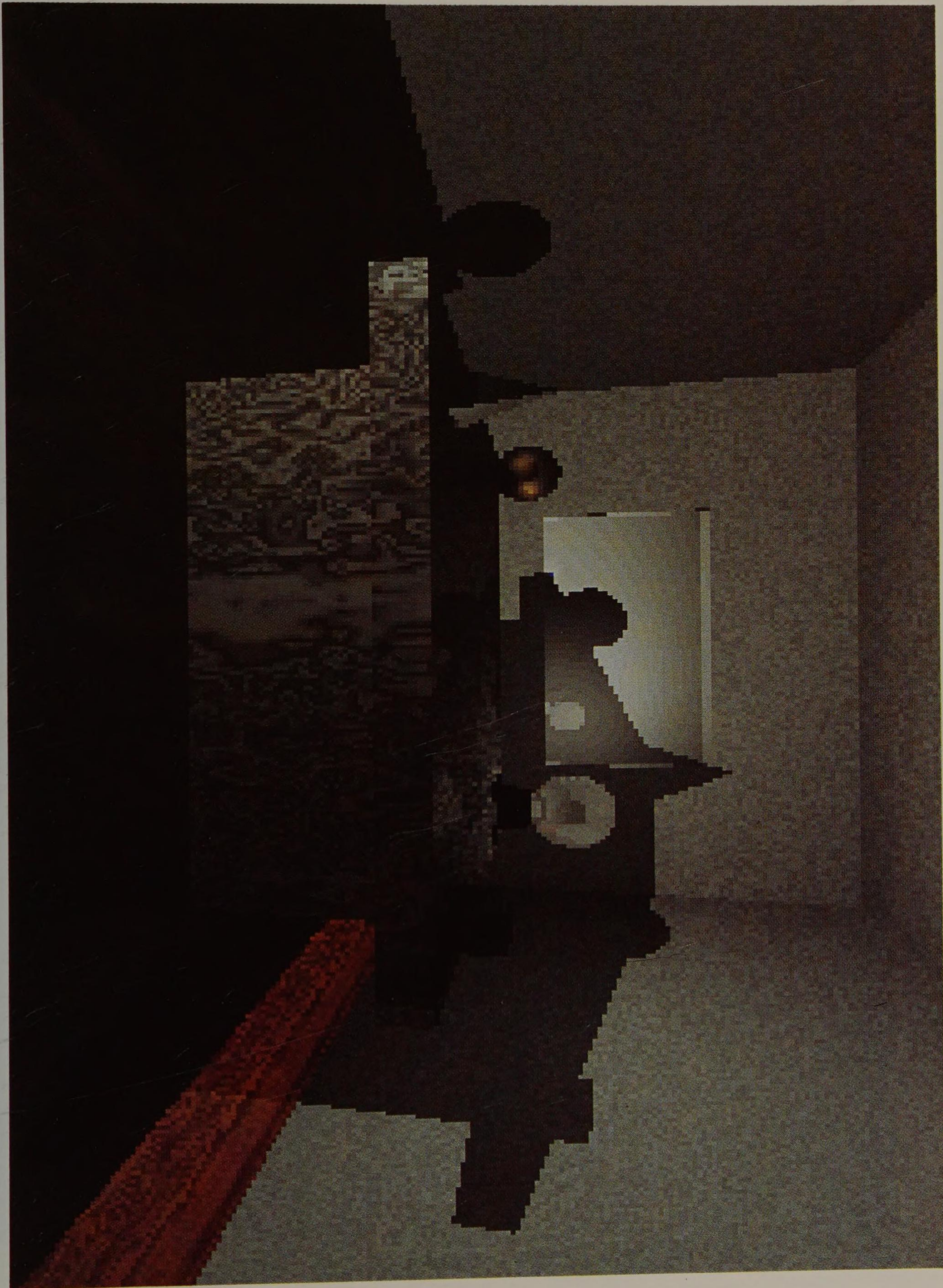
Plate 28. Ray Traced Mirrored Sphere and Light Sources

Plate 29. Ray Traced Mirrored Shapes





Plate 30. Ray Traced Marbled Spheres



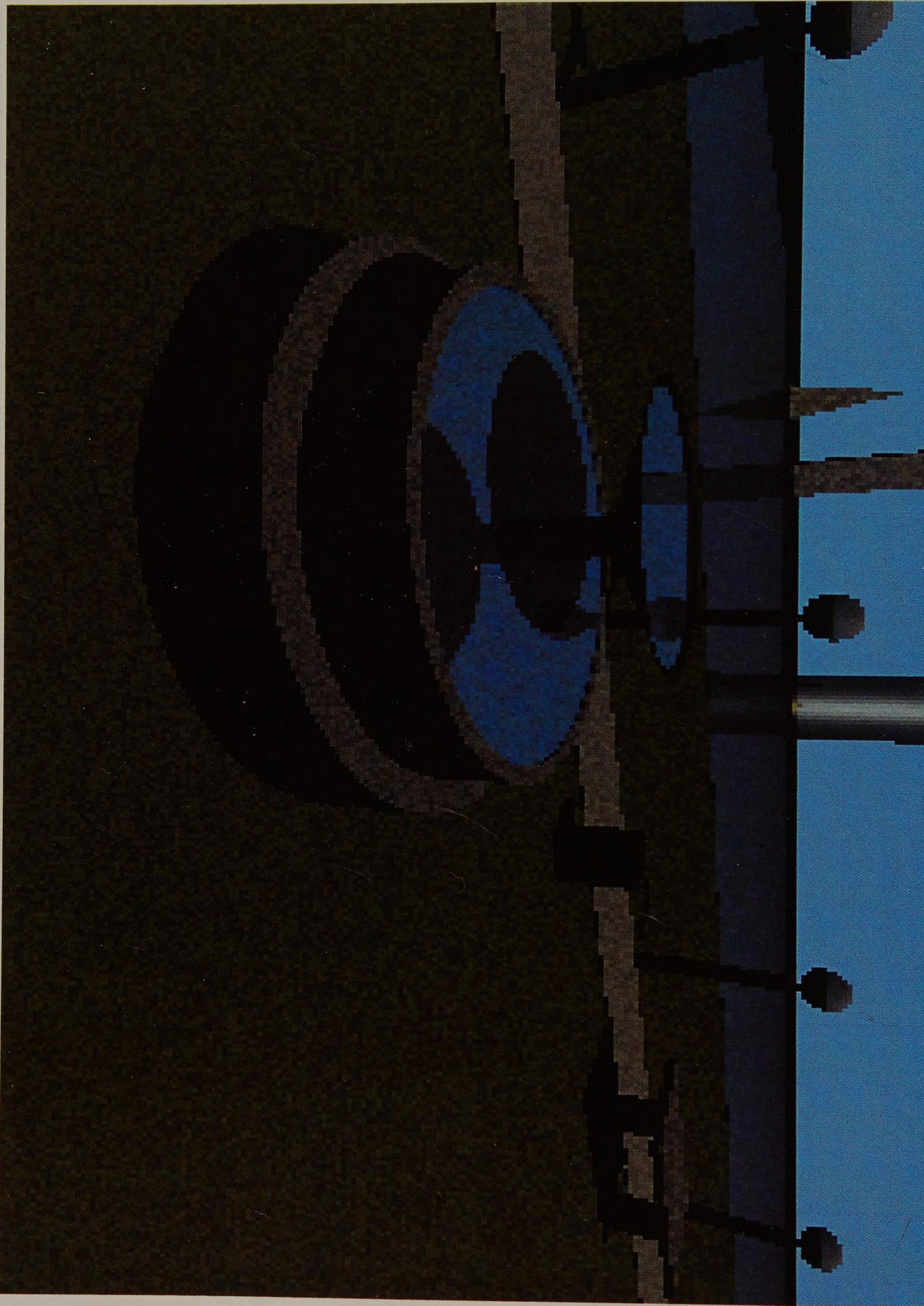


Plate 32. Ray Traced Lakeside Park

Part III

CHAPTER 11

Z Buffering Theory and Database Structure

Z BUFFERING AND HORIZON RENDERING

Z Buffering Theory and Database Structure

Z-buffering is a technique for rendering a three-dimensional scene, using a file that contains a height or z value for every point in the scene. If you display this file using x and y to define the pixel location and a color to represent the height, you would have a plasma-like display reminiscent of a contour map. To create a representation of a three-dimensional scene in two-dimensions, project each point onto the two-dimensional viewing screen. Start with the points farthest from the viewer and then process points progressing nearer to the viewer. This automatically writes near over the more distant points, yielding only that data that would actually be seen if this were a photograph. For each point, compute the color resulting from ambient light, light impinging on the object at that point, Phong shading, and reflections. The result is the color and intensity for that point. However, there is still one other problem; although you have an array of points in the original file adjacent to each other, when you project this to the viewing screen, there may be gaps between the points, consisting of pixels with no assigned color. To fix this, interpolate between the colors of the nearest points to assign colors for these pixels. The difference between this technique and the solid modeling technique is that here you have an array that defines the entire scene, whereas in the solid modeling technique, you had arrays of facets for every object in the scene.

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

{

Declare Constants and Variables

HeightBufferScalingFactor - scales height for integer
manipulation

}

Const

MaxRes = 160; { 160 x 160 pixels - actually we have memory
for 173 x 173 }

Var

Height : Array[0..MaxRes, 0..MaxRes] Of Integer;

Scaling : Integer;

MaxHeight : Integer;

Res : Integer;

Procedure HeightBufferScalingFactor;

Begin

Scaling := 32767 Div MaxHeight

End;

{

Clear, Load and Save Height Buffer Data

ClearHeightBuffer - clears all heights to Zero

SaveHeightBuffer - saves height buffer

LoadHeightBuffer - loads height buffer

GetObjectFile - get filename

GetObjectColor - text representation of color

}

Z BUFFERING THEORY AND DATABASE STRUCTURE

Type

 Name = String[32];

Var

 TextDiskFile : Text;

 ObjectFile : Name;

Procedure ClearHeightBuffer;

 Var

 I, J : Integer;

 Begin

 For I := 0 To MaxRes Do

 For J := 0 To MaxRes Do

 Height[I,J] := 0

 End;

Procedure SaveHeightBuffer(FileName : Name);

 Var

 I, J : Integer;

 Begin

 Filename := FileName + '.Dat';

 WriteLn;

 WriteLn('Saving ', FileName);

 Assign(TextDiskFile, FileName);

 Rewrite(TextDiskFile);

 WriteLn(TextDiskFile, Res, ' ', Scaling);

 For I := 0 To Res Do

 For J := 0 To Res Do

 Begin

 Write(TextDiskFile, Chr(Height[I,J] ShR 8));

 {Write MSB}

 Write(TextDiskFile, Chr(Height[I,J] And 255));

 {Write LSB}

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
        End;  
        Close(TextDiskFile)  
    End;
```

```
Procedure LoadHeightBuffer(FileName : Name);
```

```
    Var
```

```
        I, J      : Integer;  
        MSB, LSB  : Char;
```

```
    Begin
```

```
        FileName := FileName + '.Dat';  
        WriteLn;  
        WriteLn('Loading ', FileName);  
        Assign(TextDiskFile, FileName);  
        ReSet(TextDiskFile);  
        ReadLn(TextDiskFile, Res, Scaling);  
        For I := 0 To Res Do  
            For J := 0 To Res Do
```

```
                Begin
```

```
                    Read(TextDiskFile, MSB);  
                    Read(TextDiskFile, LSB);  
                    Height[I,J] := Ord(MSB) ShL 8 + Ord(LSB)
```

```
                End;
```

```
        Close(TextDiskFile)
```

```
    End;
```

```
Procedure GetObjectFile;
```

```
    Procedure MakeUpCase(Var Nm : Name);
```

```
        Var
```

```
            i : Integer;
```

```
    Begin
```

```
        For i := 1 To Length(Nm) Do  
            Nm[i] := UpCase(Nm[i])
```

```
    End;
```


Z BUFFERING THEORY AND DATABASE STRUCTURE

```
Var
  x, y : Byte;

Begin
  WriteLn;
  Write('Enter File Name => ');
  x := WhereX;
  y := WhereY;
  ReadLn(ObjectFile);
  If (ObjectFile = '') Then
    Begin
      ObjectFile := 'Mountain';
      GotoXY(x,y);
      Write(ObjectFile)
    End;
  WriteLn;
  WriteLn;
  MakeUpCase(ObjectFile)
End;

Procedure GetObjectColor(C : Integer);

Begin
  Case C Of
    0 : WriteLn('Black');
    1 : WriteLn('Blue');
    2 : WriteLn('Green');
    3 : WriteLn('Cyan');
    4 : WriteLn('Red');
    5 : WriteLn('Magenta');
    6 : WriteLn('Brown/Yellow');
    7 : WriteLn('Gray')
  End
End;
```

Figure 11-1. Listing of the *Render.Inc* File

The file that contains the variables, constants, and procedures responsible for the file and structure of the z-buffering program is called *Render.Inc*. as shown in Figure

11-1. The file begins by defining a few constants, the most important being *MaxRes*, which represents the maximum resolution of a square picture produced by z-buffering. Because you include the entire array of points defining the picture in a single array, you are limited to less than 32K points. A convenient size that will give you a little memory to spare is 160-by-160 pixels, so *MaxRes* is defined to be 160. Next, the *Height* array, which contains all the picture information, is defined. This is an array of *MaxRes*-by-*MaxRes* or 160-by-160 pixels.

Scale Factors

The first procedure in this file is *HeightBufferScalingFactor*. It defines the scaling factor for height to be 32767 divided by the maximum height. The maximum height is the greatest height of any point in the data file for a particular scene. The division is performed so that when actual height information is multiplied by the scaling factor, the resulting information will never be greater than the maximum value handled by an integer.

Clearing, Loading, and Saving Height Data

The next routines are those involved with clearing, loading, and saving the height data. The first of these procedures, *ClearHeightBuffer*, clears the *Height* buffer. It uses a pair of nested *For* loops to set every member of the *Height* array to zero.

In Chapter 14, you'll learn how to generate, examine and modify the databases for various z-buffered scenes. Here, you assume a height file already exists and you are going to load it back into the working program or, alternately, that you have already stored the data in the program and want to save it to a disk file.

The procedure, *LoadHeightBuffer* is passed the name of the data file for a z-buffered scene, not including the extension. It adds the extension *.Dat* to the file name and then displays, "Loading filename", where *filename* is the name of the desired data file. It then opens the file for reading and resets it to the beginning. Next, the procedure reads the resolution, *Res*, and the scale factor, *Scaling*. Then a pair of nested *For* loops read the pair of bytes that make up each height for the *Height* array. It converts the two bytes into an integer format. (Please observe that you will be using

a text type file, but in the interests of conserving space, you'll want vertex and facet data in the form of integers stored in two bytes, instead of a string of numbers. In the next paragraph you'll see how to store integers this way. If you look at the listing of the *LoadHeightBuffer* procedure, you'll see how to convert the two stored bytes back to an integer again.) After all height values are read, the file closes and the procedure terminates.

The *SaveHeightBuffer* procedure works just the opposite. The name of a z-buffered scene data file passes to it. The procedure first writes, "Saving filename", where *filename* is the name of the file. It opens this file as a text file for writing. Next, the *Res* and *Scaling* variables are written to the file, with a space between them. The procedure then begins a pair of nested *For* loops which iterate once for each member of the *Height* array. At each iteration, a height value converts from an integer to two bytes. The first byte is the eight most significant bits of the integer (or the most significant byte). Take the most significant byte of the integer and shift it one byte to the right to create the first character written to the file. For the second byte, the integer is anded with 255, leaving only the least significant byte, which is written to the file as the second character. When these loops are complete, the file closes and the procedure terminates.

Getting an Object File

The *GetObjectFile* procedure lets the user enter the name of the file containing height data for the desired scene. The procedure begins by displaying, "Enter File Name =>", on the screen. Next, it saves the cursor position. It then reads a line of data typed in by the user into the variable *ObjectFile*. This name appears on the screen as it is typed. If the user only hits the Enter key, the object file *Mountain* is selected by default, the cursor repositioned and this name is displayed on the screen. The procedure then calls *MakeUpCase*. This procedure simply changes the file name to all capital letters. This ends the *GetObjectFile* procedure.

Displaying the Object Color Name

The procedure *GetObjectColor* is passed a color number from 0 to 7 in the *C* parameter. It displays the name for that color (Black, Blue, etc.) on the screen.

The Render.Pas Description File Maker

When you run *Render.Pas* to generate a z-buffered scene, there are several options built into the program which are selected by the program reading a *.Des* file tailored particularly for that scene. These options are set up by the *DesMake.Pas* program before running the *Render.Pas* program. The content of this program and the files it makes are relatively simple. The program *DesMake.Pas* and the options it creates are described in this chapter. The program itself and its associated function are in the file *DesMake.Pas*, listed in Figure 12-1.

Keyboard Response Routines

First look at *Response*. This function is designed to obtain a response to a question and ensure that it is either a *Y* or an *N*. It does this by means of two nested *Repeat* loops. The inner loop continues to read a key from the keyboard until the response is other than a null, indicating that a key has been pressed. Then the keyboard character received is converted to a capital letter. If it is *Y* or *N*, the outer loop terminates; otherwise the loops iterate until an acceptable response is received. Note that whenever *Response* is called in a program, nothing can happen until you type in either *Y*, *y*, *N*, or *n*. When a correct response is received, if it is *Y*, the function returns *True*; otherwise, it returns *False*.

Now, look at *NumberResponse*. This function is very much like *Response* except that the permissible responses are the numbers from 0 to 9. These numbers are returned as ASCII characters. When an acceptable one is received, it converts to an integer, which is returned by the function.

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
{
  Program for Generation of Description Files for Render.Pas
}

Uses
  Crt;

{$I Render.Inc}

{
  Keyboard Response Routines

  Response - returns true or false to a Y or N keyboard stroke
  NumberResponse - returns a 0 to 9 based on keyboard stroke
}

Function Response : Boolean;
Var
  ch, c : Char;
Begin
  Repeat
    Repeat
      ch := ReadKey
    Until (ch<>#0);
    c := UpCase(ch);
  Until (c='Y') Or (c='N');
  WriteLn(c);
  If (c='Y') Then
    Response := True
  Else
    Response := False
End;
```


THE RENDER.PAS DESCRIPTION FILE MAKER

Function NumberResponse : Byte;

Var

c : Char;

v : Byte;

Begin

Repeat

Repeat

c := ReadKey;

Until (c<>#0);

Until (c='0') Or (c='1') Or (c='2') Or (c='3') Or (c='4')

Or (c='5') Or (c='6') Or (c='7') Or (c='8') Or (c='9');

v := Ord(c) - 48;

WriteLn(v);

NumberResponse := v

End;

{

Main Program

}

Var

NumbResp : Integer;

NDiv, i : Integer;

Bool : Boolean;

Begin

ClrScr;

WriteLn('Program to Generate a Description File for an
Object');

WriteLn;

GetObjectFile;

FileName := FileName + '.DES';

Assign(TextDiskFile, FileName);

ReWrite(TextDiskFile);

Write('Display in Half Scale? '); { Half Scale Display ? }

WriteLn(TextDiskFile, Response);

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
WriteLn;
Write('Display the Ground? ');           { Color the Ground }
Bool := Response;
WriteLn(TextDiskFile, Bool);

WriteLn;                                 { Chart of Colors }
For i := 0 To 7 Do
  Begin
    Write('Color #',i,' is ');
    GetObjectColor(i)
  End;
WriteLn;

If Bool Then
  Begin
    Write('Ground Color (1-7) = ');      { Ground Color }
    WriteLn(TextDiskFile, NumberResponse)
  End
Else
  WriteLn(TextDiskFile, 0);

WriteLn;                                { Number of Height Divisions }
Write('Number of Height Divisions = ');
NDiv := NumberResponse;
WriteLn(TextDiskFile, NDiv);
WriteLn;
WriteLn('Enter Color for each Division :');
For i := 1 To NDiv Do                    { Colors for each division }
  Begin
    Write('Color for ',i,'/', NDiv,' (1-7) = ');
    WriteLn(TextDiskFile, NumberResponse);
  End;
WriteLn;

Write('Top Color (1-7) = ');              { Top Color }
WriteLn(TextDiskFile, NumberResponse);

WriteLn;
Write('Spread of Normal Calculation = '); { Spread of
                                           Normal }
WriteLn(TextDiskFile, NumberResponse);
```


THE RENDER.PAS DESCRIPTION FILE MAKER

```
WriteLn;
WriteLn('Direction of Light');           { Light Direction }

Write('  Around the z-Axis = ');
ReadLn(NumbResp);
Write(TextDiskFile, NumbResp, ' ');

Write('  Off of the z-Axis = ');
ReadLn(NumbResp);
WriteLn(TextDiskFile, NumbResp);

Write('Tilt of the xy-Plane = ');        { Tilt of Image }
ReadLn(NumbResp);
WriteLn(TextDiskFile, NumbResp);

WriteLn;                                { Y Offset of Image }
Write('Y Offset = ');
ReadLn(NumbResp);
WriteLn(TextDiskFile, NumbResp);

Close(TextDiskFile)
End.
```

Figure 12-1. Listing of the *DesMake.Pas* File

The Main Program

The main program displays the heading “Program to Generate a Description File for an Object”, followed by a blank line. Next, *GetObject* asks for and reads in the desired file name (without an extension). The extension *.DES* is added to this file name. The file is then opened for writing. Next, the program displays, “Display in Half Scale?” and waits for a “Y” or “N” response. This is stored in the file as a Boolean *True* or *False*. If it is *True* all of the coordinates will be divided by two to display a half-scale image; otherwise, the coordinates will remain the same and a full-scale image will be created. The program then displays, “Display the Ground?” Again, a Boolean response is transmitted from the user to the file. If the response is *True*, all ground level points in the display will be in color; otherwise they will not be painted. The program then displays a chart of color names and calls *GetObjectColor* to obtain

a color number from the user. If in the above step the ground coloring option was selected, the user-entered color is stored in the file as the color for the ground and also written out to the display. If the option was not selected, a zero is stored in the file, regardless of what color the user may have chosen. The program then asks for the number of height divisions. The rendering program is set up so that the overall height range of the scene may be divided into any number of equal sections from 1 to 6. Each of these sections may then be assigned a color, and when the point on the scene being displayed is within a particular section, the selected color will be used (the shade depending upon the lighting conditions). The program next asks for the color for each division and enters it in the file. Finally, the program asks for the top color and stores the resulting entry in the file.

Next, the program asks, "Spread of Normal Calculation = ", and stores the response in the file. If the spread is zero, a single normal vector is calculated for each pixel and used in the light intensity computations. This is the mode for most scenic displays. However, in some fractal displays, the surface of the fractal object is so rough that the usual technique does not produce an acceptable surface. For such scenes, a number may be entered for the spread, and that number of additional normals, surrounding the pixel in each direction, will be computed and averaged to obtain the normal value, giving a smoother surface. The program now turns its attention to the light direction, asks for and records the direction of the light source around the z -axis and off the z -axis. The tilt of the x - y plane is then requested and stored in the file. Then the y offset is requested and stored in the file. These last two terms effectively define the viewing position of the scene. This completes the description file and the program terminates.

A number of description and scene data files are furnished as part of the program disk. The contents of the description files and the plate or figure number where the scene is shown in the book are given in Figure 12-2. How to make up the files of scene data is described in Chapter 14.

THE RENDER.PAS DESCRIPTION FILE MAKER

Item	HmSphere	Bowl	HmToroid	PlotEqn1
Plate Number	22	19		24
Half Scale?	Y	Y	Y	N
Display Grnd?	Y	Y	Y	Y
Ground Color	1	1	1	2
No. of Ht. Div	5	5	5	1
Color for Div 1	2	2	2	2
Color for Div 2	3	3	3	
Color for Div 3	4	4	4	
Color for Div 4	5	5	5	
Color for Div 5	6	6	6	
Color for Div 6				
Top Color	7	7	7	2
Normal Spread	0	0	0	0
Light Direction around z-axis	=30	-30	-30	45
Light Direction off z-axis	45	45	45	60
Tilt of xy-plane	35	35	35	23
Y offset	60	0	0	0

Figure 12-2. Description File Contents

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

Item	PlotEqn2	PlotEqn3	PlotEqn4	MagnFlds
Plate Number	25	26	27	23
Half Scale?	N	N	N	Y
Display Grnd?	Y	Y	Y	Y
Ground Color	4	6	5	1
No. of Ht. Div	1	1	1	5
Color for Div 1	4	6	5	2
Color for Div 2				3
Color for Div 3				4
Color for Div 4				5
Color for Div 5				6
Color for Div 6				
Top Color	4	6	5	7
Normal Spread	0	0	0	0
Light Direction around z-axis	45	45	45	45
Light Direction off z-axis	60	60	23	45
Tilt of xy-plane	23	23	0	35
Y offset	0	0		0

Figure 12-2. Description File Contents (continued)

THE RENDER.PAS DESCRIPTION FILE MAKER

Item	Mountain	Mountan1	Mountan2	CPM1
Plate Number	Fig. 15-2	Fig. 15-3	Fig. 15-4	20
Half Scale?	N	N	N	N
Display Grnd?	Y	Y	Y	Y
Ground Color	7	7	7	0
No. of Ht. Div	1	1	1	1
Color for Div 1	7	7	7	1
Color for Div 2				
Color for Div 3				
Color for Div 4				
Color for Div 5				
Color for Div 6				
Top Color	7	7	7	0
Normal Spread	0	0	0	0
Light Direction				
around z-axis	-35	-35	-35	-35
Light Direction				
off z-axis	60	60	60	40
Tilt of xy-plane	30	30	90	18
Y offset	0	0	0	70

Figure 12-2. Description File Contents (continued)

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

Item	CPM2	CPMJ1	CPMJ2	Dragon	RevSolid
Plate Number		21			
Half Scale?	N	N	N	Y	Y
Display Grnd?	Y	Y	Y	N	N
Ground Color	0	0	0	0	0
No. of Ht. Div	1	1	1	1	1
Color for Div 1	5	3	1	4	2
Color for Div 2					
Color for Div 3					
Color for Div 4					
Color for Div 5					
Color for Div 6					
Top Color	7	7	5	4	2
Normal Spread	0	0	0	1	1
Light Direction					
around z-axis	-144	-35	-35	0	160
Light Direction					
off z-axis	70	40	40	75	65
Tilt of xy-plane	18	22	22	90	90
Y offset	140	70	70	0	0

Figure 12-2. Description File Contents (continued)

The Z-Buffer Rendering Program

Before going into the details of how to create databases rendered by the z-buffering technique, start by looking at the details of the program itself. Assume that you have an array of data, contained in a two-dimensional 160-by-160 array, where each array position, represents a particular x and y position and the contents of that member of the array represents the height of the scene at that point. The program to perform the rendering operation is called *Render.Pas*, as shown in Figure 13-1. The program and procedures are described in the following paragraphs.

Screen Procedures

This section is concerned with ensuring that each pixel on the screen is displayed with proper color and intensity. If you are in the half-scale mode, this is relatively simple. Looking at the procedure *Pixel*, you will note that in the half-scale mode, when the procedure is passed a pair of coordinates (x,y), it simply takes these as the coordinates of the pixel. It reverses the y value so that increasing values of y move upward on the screen, rather than the perverse usual mapping of the screen that makes y increase in a downward direction. The intensity also passes the procedure and using that and the current color, the procedure calls *PutPixel*, described in Chapter 2, to paint the pixel on the screen with the proper shade of color. The process, however, becomes more complicated for the full-scale mode, because there are points on the screen between data points of the z-buffered array, so that interpolation is required. If you really want to know exactly how this works, you'll have to trace every step of the code, which is tedious and convoluted. What follows is an overall picture.

There are three buffers called *PixBufColor* for storing pixel color and three called *PixBufInten* for storing pixel intensity. In the full-scale mode, instead of writing pixel data to the screen, information is stored in one pair of these buffers, until a full column of picture data accumulates. The buffers rotate so that data for the two latest columns is always available. Each time a new column of data has accumulated, the procedure calls *InterpolatePixBuf* to interpolate. The Interpolation procedures determine color and intensity for each pixel written to the screen at locations between data points. It does this by randomly selecting between colors and intensities of adjacent data points from data stored in the buffer pairs. The resulting data for a column is then displayed on the screen. The procedures involved in this process are *CopyPixBuf*, *InterpolatePixBuf*, *InterpolateY*, *InterpolateX*, and *DisplayPixBuf*.

```
{
```

z-buffer Rendering Package

The data is in an integer valued (MaxRes+1 x MaxRes+1 x 32768) cube where an integer height value is stored in a cell at (x,y). The height value is then scaled to fit into the unit cube.

tilt = 0 for side view and tilt = 90 for top view
 looking from top view : x-axis is right while y-axis is up
 looking from side view : x-axis is right while y-axis is in

```
}
```

Uses

Dos, Crt;

```
{ $I Math.Inc }
```

```
{ $I Graph.Inc }
```

```
{ $I Render.Inc }
```


THE Z-BUFFER RENDERING PROGRAM

Screen Procedures

{

Pixel - the place interpolated pixels routine

}

Const

MaxBufs = 3;

Type

PixBuffer = Array[0..(MaxRes * 2)] Of Byte;

PixBuf = Array[1..MaxBufs] Of PixBuffer;

Var

PixBufColor : PixBuf;

PixBufInten : PixBuf;

NullPixBuf : PixBuf;

TwoMaxRes : Integer;

Procedure ClearPixBuf(Buf1, Buf2 : Integer);

Var

i : Byte;

Begin

For i := Buf1 To Buf2 Do

Begin

PixBufColor[i] := NullPixBuf[i];

PixBufInten[i] := NullPixBuf[i]

End

End;

Procedure CopyPixBuf(Buf1, Buf2 : Integer);

Begin

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
    PixBufColor[Buf2] := PixBufColor[Buf1];  
    PixBufInten[Buf2] := PixBufInten[Buf1]  
End;
```

Procedure InterpolatePixBuf;

Procedure InterpolateX(Buf, Buf1, Buf2 : Integer);

```
    Var  
        i : Integer;
```

Begin

```
    For i := 0 To TwoMaxRes Do
```

```
        Begin
```

```
            If (Random(2)=1) Then
```

```
                PixBufColor[Buf][i] := PixBufColor[Buf1][i]
```

```
            Else
```

```
                PixBufColor[Buf][i] := PixBufColor[Buf2][i];
```

```
                PixBufInten[Buf][i] := ( PixBufInten[Buf1][i] +  
                                         PixBufInten[Buf2][i] ) Div 2
```

```
            End
```

```
        End;
```

Procedure InterpolateY(Buf : Integer);

```
    Var
```

```
        i, k : Integer;
```

Begin

```
    For i := 0 To MaxRes-1 Do
```

```
        Begin
```

```
            k := i * 2;
```

```
            If (Random(2)=1) Then
```

```
                PixBufColor[Buf][k + 1] := PixBufColor[Buf][k]
```

```
            Else
```

```
                PixBufColor[Buf][k + 1] := PixBufColor[Buf][k +  
                                                         2];
```

```
                PixBufInten[Buf][k + 1] :=( PixBufInten[Buf][k]  
                                              + PixBufInten[Buf][k + 2] ) Div 2
```

```
            End
```


THE Z-BUFFER RENDERING PROGRAM

End;

Begin
InterpolateY(1);
InterpolateY(3);
InterpolateX(2,1,3)
End;

Var
OldCol : Integer;
YOffset : Integer;
HalfScale : Boolean;
ShowGround : Boolean;

Procedure DisplayPixBuf(Buf1, Buf2 : Integer);

Var
Buf, Row, x, yc : Integer;

Begin
x := (OldCol - 1) * 2 - 1;
yc := MaxY + YOffset;
For Buf := Buf1 To Buf2 Do
For Row := 0 To TwoMaxRes Do
If Not(PixBufColor[Buf][Row] = 0) Then
PutPixel((Buf + x), (yc - Row),
PixBufColor[Buf][Row], PixBufInten[Buf][Row])
End;

Var
ColNum : Integer;
RowNum : Integer;
BufNum : Byte;
MaxBuf : Byte;
FoundFirstCol : Boolean;
DisplayColor : Byte;

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
Procedure Pixel(X, Y : Integer;      Inten : Byte);
Begin
  If HalfScale Then
    PutPixel(X, 199-Y, DisplayColor, Inten)
  Else
    Begin
      ColNum := X;
      RowNum := 2 * Y;

      If ShowGround Then
        Begin
          If (ColNum > 0) Then
            BufNum := MaxBuf
          Else
            If (ColNum = 0) Then
              BufNum := 1
            End
          Else
            If FoundFirstCol Then
              BufNum := MaxBuf
            Else
              If (ColNum > 0) Then
                Begin
                  OldCol := ColNum;
                  BufNum := 1;
                  FoundFirstCol := True
                End;
              If (ColNum = 2) Then
                If (OldCol = 1) Then
                  Begin
                    InterpolatePixBuf;
                    DisplayPixBuf(1,MaxBuf); {interpolate
                                                columns 0 and 1}
                    OldCol := ColNum;
                    CopyPixBuf(MaxBuf,1);
                    ClearPixBuf(2,MaxBuf);
                    Exit
                  End;
                End;
            End;
        End;
    End;
```


THE Z-BUFFER RENDERING PROGRAM

```
If (OldCol = ColNum-1) Then
```

```
  Begin
```

```
    InterpolatePixBuf;
```

```
    DisplayPixBuf(2,MaxBuf); { interpolate  
                             columns n and n-1 }
```

```
    OldCol := ColNum;
```

```
    CopyPixBuf(MaxBuf,1);
```

```
    ClearPixBuf(2,MaxBuf);
```

```
    Exit
```

```
  End;
```

```
  PixBufColor[BufNum][RowNum] := DisplayColor;
```

```
  PixBufInten[BufNum][RowNum] := Inten
```

```
End
```

```
End;
```

```
{
```

The Illumination Model

Intensity - calculate the intensity of a pixel

```
}
```

```
Var
```

```
  Spread : Integer;
```

```
  Light : TDA;
```

```
  View : TDA;
```

```
Function Intensity(i, j : Integer) : Integer;
```

```
  Const
```

```
    Ambient = 0.15; { percentage of ambient light }
```

```
    DifRfl = 0.45; { percentage of diffuse light from  
                  reflection }
```

```
    SpcRfl = 0.40; { percentage of specular light from  
                  reflection }
```

```
    Gloss = 2; { shininess }
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```

Var
  CosTheta : Real; { cosine of angle between 'n' and 'l' }
  CosAlpha : Real; { cosine of angle between 'v' and 'r' }
  SrfNorm   : TDA;
  Ref       : TDA;
  SrfNormSum : TDA;
  dx, dy, dz : Real;
  Cells      : Integer;
  i1, j1     : Integer;
  TwoCosTheta : Real;
  Temp       : TDA;

Begin
  VecNull(SrfNormSum);
  Cells := Sqr(2 * Spread + 1);
  For i1 := (i-Spread) to (i+Spread) Do
    For j1 := (j-Spread) to (j+Spread) Do
      Begin
        If (j1-1 < 0) Then
          Vec(0,0,1, SrfNorm)
        Else
          Begin
            dx := ( Height[i1, j1] - Height[i1+1, j1] ) /
                  Scaling;
            dy := ( Height[i1, j1-1] - Height[i1, j1] ) /
                  Scaling;
            dz := 1 / (Res-1);
            Vec(dx, dy, dz, SrfNorm);
            VecNormalize(SrfNorm)
          End;
        VecAdd(SrfNormSum, SrfNorm, SrfNormSum);
      End;
  VecScalMult(1 / Cells, SrfNormSum, SrfNorm);
  CosTheta := VecDot(SrfNorm, Light); { cos  $\Theta$  = n·l }
  If (CosTheta < 0) Then
    Intensity := Round( MaxInten * Ambient )
  Else
    Begin
      TwoCosTheta := 2 * CosTheta;
      VecScalMult(TwoCosTheta, SrfNorm, Temp);
    End;

```


THE Z-BUFFER RENDERING PROGRAM

```

VecNormalize(Temp);
VecSub(Temp, Light, Ref);
VecNormalize(Ref);
CosAlpha := VecDot(View, Ref); {  $\cos \alpha = v \cdot r$  }
Intensity := Round( MaxInten * ( Ambient + DifRfl*
CosTheta + SpcRfl * Power( CosAlpha , Gloss ) ) )

```

```

End
End;

```

```
{
```

Load Description Data

```

LoadDescription - loads description of object

```

```
}
```

```
Const
```

```
    MaxDiv = 6;
```

```
Var
```

```

NumHgtDiv    : Integer;
GroundColor  : Integer;
HeightColor  : Array[1..MaxDiv] Of Integer; { 6 Height
                                              Divisions => 6 Colors }

TopColor     : Integer;
HeightLimit  : Array[1..MaxDiv] Of Integer;
SinViewPhi   : Real;
CosViewPhi   : Real;
Phi, Theta   : Real;
x, y, z      : Real;
LightPhi     : Real;
LightTheta   : Real;

```

```
Procedure LoadDescription(FileName : Name);
```

```
    Type
```

```
        Strg = String[5];
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
Var
  B      : Strg;
  I1, I2 : Integer;
  i      : Integer;

Function Bool(B : Strg) : Boolean;

  Begin
    If (B = 'FALSE') Then
      Bool := False
    Else
      Bool := True
    End;

Begin
  FileName := FileName + '.DES';
  Assign(TextDiskFile, FileName);
  Reset(TextDiskFile);

  WriteLn;
  ReadLn(TextDiskFile, B);      { Make Image Half Scale ? }
  If Bool(B) Then
    Begin
      HalfScale := True;
      WriteLn('Scale = Half');
    End
  Else
    Begin
      HalfScale := False;
      WriteLn('Scale = Full');
    End;
  WriteLn;
  ReadLn(TextDiskFile, B);      { Show Ground or leave Black }
  If Bool(B) Then
    ShowGround := True
  Else
    Begin
      ShowGround := False;
      WriteLn('Ground Coloring Off');
    End;
End;
```


THE Z-BUFFER RENDERING PROGRAM

```
ReadLn(TextDiskFile, I1);      { Ground Color }
GroundColor := I1;
If ShowGround Then
  Begin
    Write('Ground Color = ');
    GetObjectColor(I1)
  End;

ReadLn(TextDiskFile, NumHgtDiv); { Get Number of Height
                                   Divisions }

For i := 1 To NumHgtDiv Do      { Get Colors for Height
                                   Divisions }
  Begin
    ReadLn(TextDiskFile, I2);
    HeightColor[i] := I2;
    Write('Color below ', i, '/', NumHgtDiv, ' = ');
    GetObjectColor(I2)
  End;

ReadLn(TextDiskFile, I1);      { Top Color }
TopColor := I1;
Write('Top Color = ');
GetObjectColor(I1);

WriteLn;
ReadLn(TextDiskFile, I1); { Spread of Normal Calculation }
InitSpread(I1);
WriteLn('Normal Calculation Spread = ', I1);

WriteLn;
ReadLn(TextDiskFile, I1, I2); { Light Direction }
LightPhi := I1;
LightTheta := I2;
WriteLn('Light Direction is ', I1:4, '°' around the z-Axis
        and');
WriteLn('      ', I2:4, '°' off of the z-Axis');

WriteLn;
ReadLn(TextDiskFile, I1);      { Tilt of XY Plane }
```



```

Phi := Radians(I1);
SinViewPhi := Sin(Phi);
CosViewPhi := Cos(Phi);
x := 0;
y := -CosViewPhi;
z := SinViewPhi;
Vec(x, y, z, View);
Phi := Radians( LightPhi );
Theta := Radians( LightTheta );
x := Sin( Theta ) * Cos( Phi );
y := Sin( Theta ) * Sin( Phi );
z := Cos( Theta );
Vec(x, y, z, Light);
WriteLn('Tilt of xy-Plane = ',I1:4,'°');

WriteLn;
ReadLn(TextDiskFile, I1);          { Y Offset of Image }
WriteLn('Y Offset = ',I1:4);
YOffset := I1;
Close(TextDiskFile)
End;

```

```
{
```

Display the Height Field

DisplayHeightField - calculate pixel positions and interpolate intensities

```
}
```

```
Var
```

```
Max : Integer;
```

```
Procedure DisplayHeightField;
```

```
Function Proj(y, z : Real) : Integer;
```


THE Z-BUFFER RENDERING PROGRAM

```
Begin
  Proj := Round( Res * ( y * SinViewPhi + z * CosViewPhi))
End;
```

```
Var
  i, j      : Integer;
  p0, p1    : Integer;
  horizon   : Integer;
  Pix       : Integer;
```

```
Procedure AboveHorizon;
```

```
Var
  OldPix : Integer;
  NewPix : Integer;
  p      : Integer;
```

```
Procedure Interpolate;
```

```
Var
  h : Real;
```

```
Begin
  h := (p - p0) / (p1 - p0);
  Pix := Round( ( h * OldPix + (1 - h) * NewPix ) );
  Pixel(i, p, Pix);
  p := p - 1
End;
```

```
Begin
  OldPix := Intensity(i, j);
  Pixel(i, p1, OldPix);
  p := p1 - 1;
  If ShowGround And (p > horizon) Then { interpolate - if
                                         points are }
  Begin                                { left out between old and new }
    NewPix := Intensity(i, j-1); { horizons and there
                                   is ground }
    While (p > horizon) Do Interpolate
  End;
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```

        horizon := p1
    End;

Var
    n : Integer;

Label 1, 2;

Begin
    For n := 1 To NumHgtDiv Do
        HeightLimit[n] := n * (Max Div NumHgtDiv);
        For i := (0 + Spread) To ((Res-1) - Spread) Do
            Begin
                p0 := Proj(0, Height[i,0] / Scaling);
                horizon := p0;
                For j := (1 + Spread) To ((Res-1) - Spread) Do
                    Begin
                        If (Height[i,j] = 0) Then
                            If ShowGround Then
                                DisplayColor := GroundColor
                            Else
                                Goto 1
                        Else
                            Begin
                                For n := 1 To NumHgtDiv Do
                                    If (Height[i,j] < HeightLimit[n]) Then
                                        Begin
                                            DisplayColor := HeightColor[n];
                                            Goto 2
                                        End;
                                DisplayColor := TopColor; { Height
                                                                must be Max so TopColor }
                            End;
                        2:
                            End;
                        p1 := Proj(j / (Res-1), Height[i,j] / Scaling);
                        If (p1 > horizon) Then AboveHorizon;
                        p0 := p1;
                    End
                1:
                    End
            End
        End;
    End;
End;

```


THE Z-BUFFER RENDERING PROGRAM

Procedure FindMax;

Var

i, j : Integer;

Begin

Max := 0;

For i := 0 To MaxRes Do

For j := 0 To MaxRes Do

If (Height[i,j] > Max) Then Max := Height[i,j]
{ find max height }

End;

{

Place the Special Add-Ons To the Screen

PlaceAdd-OnsToScreen - special additions such as lightning and moons

}

{\$I Add-Ons.Inc}

Procedure PlaceAdd-OnsToScreen(ObjectFile : Name);

Begin

If (ObjectFile = 'CPM1') Then

Begin

JuliaSet(0.0, 1.0, 0, Res Div 3, Res, Res, Blue);

MapJuliaSetToSphere(-0.15652, -1.03225, -3*Res Div 4,
-Res Div 2, 35, 90, 90, Magenta)

End;

If (ObjectFile = 'CPM2') Then

Begin

JuliaSet(0.0, 1.0, 0, Res Div 3, Res, Res, Red);

MapJuliaSetToSphere(-0.15652, -1.03225, -3*Res Div 4,
-Res Div 2, 35, 90, 90, Blue)

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

End;

If (ObjectFile = 'CPMJ1') Then

Begin

JuliaSet(0.0, 1.0, 0, Res Div 3, Res, Res, LightGray);

MapJuliaSetToSphere(-0.15652, -1.03225, -3*Res Div 4,
-Res Div 2, 35, 90, 90, Yellow)

End;

If (ObjectFile = 'CPMJ2') Then

Begin

Sky(0, 0, Res, Res, Cyan);

MapJuliaSetToSphere(-0.15652, -1.03225, -3*Res Div 4,
-Res Div 2, 35, 90, 90, Magenta)

End;

End;

{

Main Program

}

Var

i, j : Integer;

Begin

Title;

WriteLn('z-Buffer Rendering System');

WriteLn;

ClearHeightBuffer;

GetObjectFile;

LoadDescription(ObjectFile);

Randomize;

OldCol := 1;

TwoMaxRes := 2 * MaxRes;

FoundFirstCol := False;

For i := 1 To MaxBufs Do

THE Z-BUFFER RENDERING PROGRAM

```
For j := 0 To TwoMaxRes Do
    NullPixBuf[i][j] := 0;
If HalfScale Then
    MaxBuf := 1
Else
    MaxBuf := MaxBufs;
ClearPixBuf(1,MaxBufs);

LoadHeightBuffer(ObjectFile);

InitGraphics;
InitPalette2(Color);
SetPalette(Color);

FindMax;

PlaceAdd-OnsToScreen(ObjectFile);

DisplayHeightField;
ExitGraphics
End.
```

Figure 13-1. Listing of *Render.Pas* File

The Illumination Model

The illumination model determines the intensity of color at any given data point. This is done by the function *Intensity*. The parameters *i* and *j* to this function are the *x* and *y* coordinates of the point whose light intensity is being determined. The function zeroes *SrfNormSum*, which stores the surface normal vector summation for the designated point. Next, the function looks at *Spread*, which tells how many adjacent points are used in computing the surface normal vector. For each point, the function computes the coordinates of the surface normal vector and adds that vector to *SrfNormSum*. When all of the normals have been computed, the sum is divided by the number of points used to obtain an average, which is *SrfNorm*, the surface normal vector.

Next, the function finds the angle *theta* between the light source and the surface normal vector. If the cosine of this angle is less than zero, none of the light from the

light source is reflected from the point, so the intensity is set to that of ambient light. Otherwise, the function determines the Phong shading reflection, found by raising the cosine of alpha to the power defined for the glossiness. This cosine is the angle between the viewer position vector and a reference vector which is the normalized difference between the light source vector and the angle at which light is reflected off of the point. The function then determines the light intensity at the point, which is the sum of the ambient light, the diffused light factor multiplied by the cosine of theta, and the specular reflection factor multiplied by the cosine of alpha raised to the glossiness power.

Loading the Description Data

Before you can do the rendering process, you need to load data from the description file to set up the rendering parameters. These parameters are described in the previous chapter. Here is a description of how they are transferred to the data base for the rendering program.

The procedure *LoadDescription* performs the loading function. Before it is listed, *Bool* is listed. This function returns a Boolean variable which is *False* if the string "FALSE" is passed to the function and *True* otherwise. The procedure *LoadDescription* is passed a filename as a parameter. It appends the extension *.DES* to this file name, it then opens and resets the file, then reads the scale parameter from the file. If the parameter is TRUE the procedure sets *HalfScale* to true and writes "Scale=Half" to the screen. Otherwise it sets *HalfScale* to false and writes "Scale=Full" to the screen. Next the procedure reads and converts to Boolean the parameter that determines if the ground is to be colored. It sets the parameter *ShowGround* to this Boolean value, and if false, displays "Ground Coloring Off" on the screen. Next, the procedure reads the number of height divisions from the file and enters this value into *NumHgtDiv*. The procedure then enters a loop which iterates for the number of specified height divisions, reading the color for each from the file, saving it in the proper member of the array *HeightColor* and then displaying the color on the screen. The top color is read from the file, saved in *TopColor* and displayed on the screen. In a similar manner, the parameters *Spread*, *LightPhi*, and *LightTheta* are read and displayed. The tilt of the xy plane is read as an angle in degrees, converted to radians,

and the sine and cosine of the angle found and stored. These are used to find the *View* vector. The light vector is computed using the light and view information. Finally, the procedure reads the *y* offset from the file, saves it in *YOffset* and displays it on the screen.

The *Render.Pas* Program

You're not quite through all the associated procedures yet, but why not try an overview of the main rendering program, *Render.Pas*. This program begins by clearing the height buffer. It then gets the name of the desired data file from the user. It calls *LoadDescription* to obtain the various parameter values from the *.DES* file and zeroes out all of the pixel intensity and color buffers, and calls *LoadHeightBuffer* to get all of the picture height information from the appropriate *.DAT* file. Next, *PlaceAdd-OnsToScreen* is called with the file name as a parameter to permit any added displays to be painted to the screen. When this is complete, the procedure *DisplayHeightField* displays the main scene. Finally the program exits from the graphics mode and terminates.

Displaying the Height Field

You've just learned that a scene can be made up of special effects called *AddOns* and a main, three-dimensional scene. In this section, the procedure *DisplayHeightField*, which converts the z-buffer array to a depiction of a three-dimensional scene is described. There will be more about *Add-Ons* in the next section. (Note: Most of the z-buffered pictures do not have any *Add-Ons*.) The *DisplayHeightField* procedure determines from the parameter *NumHgtDiv*, the number of specified height divisions and the height for each division. The procedure next enters a *For* loop which iterates once for each column of the display. At the start of each iteration, the procedure calls *Proj*, which computes the proper row for the projection of the initial value of *y* and its corresponding height value onto the two-dimensional screen. The parameter *horizon*, is set to this value. Next an inner *For* loop iterates once for each row in the column. If the height for the current column and row is zero and *ShowGround* is false, the loop proceeds to the next iteration; if *ShowGround* is true, the parameter *DisplayColor* is set to the ground color. Alter-

nately, if the height is greater than zero, the procedure determines which height division the height is in, and sets *DisplayColor*, accordingly. The procedure then calls *Proj*, which computes the proper row for the projection of the current value of *y* and its corresponding height value onto the two-dimensional screen. The geometry is shown in Figure 13-2. A *y-z* plane slice of the three-dimensional space is projected onto the screen as a single line having the maximum height of the object for that slice. If this value is greater than the value of *horizon*, the procedure calls *AboveHorizon*. This procedure paints a pixel on the screen, and if there were pixels left out between the current pixel and the last pixel painted to the screen, it interpolates to find the intensity of the intermediate pixels and paints them to the screen too. It stores the value of the current pixel as *OldPix* for reference. The two *For* loops in *DisplayHeightField* iterate until all the pixels have been painted on the screen.

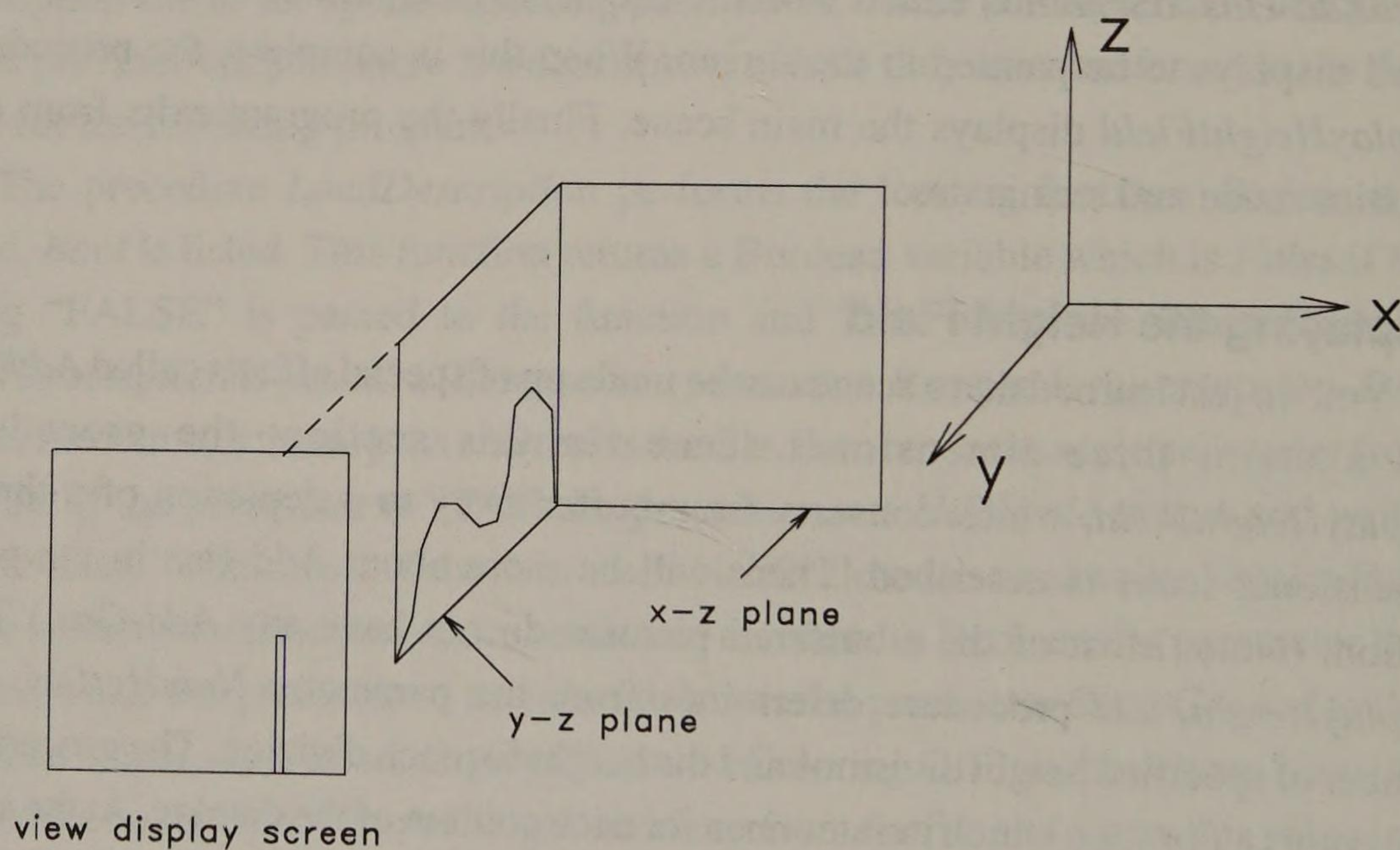


Figure 13-2. Geometry for Z-Buffer Projection

Placing Add-Ons on the Screen

If you look at the procedure *PlaceAddOnsToScreen*, note that it is only operative for four of the data files that create z-buffered pictures. These are *CPM1*, *CPM2*,

CPMJ1, and *CPMJ2*. Certain procedures create specialized displays for each of these files. These specialized procedures are included in the file *AddOns.Inc*, listed in Figure 13-3.

Julia Set Images

The Julia set is a plot of the iterated equation:

$$z_n = z_{n-1}^2 + c \quad (\text{Equation 13-1})$$

where c is a complex number held fixed for a particular Julia set while the complex number z_0 (the initial value of z) varies over the range of the screen, with the real part plotted horizontally and the imaginary part, vertically. Figure 13-4 is a chart of C values with a description of Julia set that results for each. Now look at procedure *Scan*. It determines the incremental values of x and y for the specified display resolution, then enters a pair of nested *For* loops which iterate over the whole range of x and y for the desired display. At each iteration of the inner loop, the initial values of x and y (the real and imaginary parts of z_0 in the equation above). The procedure then calls *Iterate* to perform the actual iteration of the equation. This procedure repeats the iteration until the sum of the squares of x and y is greater than 100. When this occurs, the *Inten* parameter is set to the number of iterations and the procedure terminates. If the number of iterations becomes equal to the maximum intensity level without the above condition occurring, the procedure sets *Inten* to zero and the procedure terminates. Back in the *Scan* procedure, the intensity is increased by 5. For an ordinary Julia set at half-scale display, this intensity is then plotted to the screen in the designated location and color. For an ordinary Julia set at full-scale, the intensity is plotted to four pixels which form a square.

Special Effects for z-Buffered Scene Creation

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
JuliaSet - plot a Julia set ( lightning )
MapJuliaSetToSphere - map a Julia set onto a sphere (moon with
                    dendrite)
Sky      - blend a sky color from the horizon to the zenith
}

```

```
Procedure PlotSphrPix(LocX, LocY, x, y, R : Integer; Color,
    Intensity : Byte);
```

```
Procedure PutDitheredSphere(xx, yy : Integer);
```

```
Var
```

```
  Xc, Yc, Zc : Real;
```

```
  Xp, Yp      : Integer;
```

```
  Theta, Phi  : Real;
```

```
Begin
```

```
  Theta := Radians( yy );
```

```
  Phi   := Radians( xx );
```

```
  Xc := R * Sin( Theta ) * Cos( Phi );
```

```
  Yc := R * Sin( Theta ) * Sin( Phi );
```

```
  Zc := R * Cos( Theta );
```

```
  MapCoordinates( Xc, Yc, Zc, Xp, Yp );
```

```
  PutPixel( Xp + LocX, Yp + LocY, Color, Intensity )
```

```
End;
```

```
Begin
```

```
  x := 2 * x;
```

```
  y := 2 * y;
```

```
  PutDitheredSphere( x , y );
```

```
  PutDitheredSphere( x+1, y );
```

```
  PutDitheredSphere( x , y+1 );
```

```
  PutDitheredSphere( x+1, y+1 )
```

```
End;
```

```
{
```


THE Z-BUFFER RENDERING PROGRAM

Julia Set Images

JuliaSet - plot a Julia set
MapJuliaSetToSphere - map a Julia set onto a sphere

}

Procedure Iterate(p, q, xo, yo : Real; Var Inten : Integer);

Const

MaxIntens = (MaxInten-5); { Number of Intensities }
DivBndSqr = 100; { Divergence Boundary Squared }

Var

xn, yn : Real;
Iters : Integer;

Label 1;

Begin

Iters := 0;
1: xn := Sqr(xo) - Sqr(yo) + p;
yn := 2 * xo * yo + q;
xo := xn;
yo := yn;
Iters := Iters + 1;
If ((Sqr(xn) + Sqr(yn)) > DivBndSqr) Then
Begin
Inten := Iters;
Exit
End;
If (Iters = MaxIntens) Then
Begin
Inten := 0;
Exit
End;
Goto 1
End;

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

Var

SphrPix : Boolean;

Procedure Scan(p, q, xmin, ymin, xmax, ymax : Real;
LocX, LocY, Rad, ResX, ResY, Color : Integer);

Var

dx, dy : Real;

x, y : Integer;

xo, yo : Real;

Inten : Integer;

Begin

dx := (xmax-xmin) / (ResX-1);

dy := (ymax-ymin) / (ResY-1);

For x := 0 To (ResX-1) Div 2 Do

For y := 0 To (ResY-1) Do

Begin

xo := xmin + x * dx;

yo := ymin + y * dy;

Iterate(p, q, xo, yo, Inten);

Inten := Inten + 5;

If SphrPix Then

Begin

If HalfScale Then

Begin

PlotSphrPix(LocX, 3*Res Div 8 + LocY,
x, y, Rad, Color, Inten);

PlotSphrPix(LocX, 3*Res Div 8 + LocY,
ResX-1-x, ResY-1-y, Rad, Color, Inten)

End

Else

Begin

PlotSphrPix(LocX, LocY, x, y, Rad,
Color, Inten);

PlotSphrPix(LocX, LocY, ResX-1-x,
ResY-1-y, Rad, Color, Inten)

End

End

THE Z-BUFFER RENDERING PROGRAM

```

Else
  Begin
    If HalfScale Then
      Begin
        PutPixel( LocX + x, LocY + y, Color,
                  Inten );
        PutPixel( LocX + ResX-1-x, LocY +
                  ResY-1-y, Color, Inten )
      End
    Else
      Begin
        PutPixel( LocX + x*2 , LocY + y,
                  Color, Inten );
        PutPixel( LocX + x*2+1, LocY + y,
                  Color, Inten );
        PutPixel( LocX + (ResX-1-x)*2 , LocY +
                  ResY-1-y, Color, Inten );
        PutPixel( LocX + (ResX-1-x)*2+1, LocY
                  + ResY-1-y, Color, Inten )
      End
    End
  End
End;

```

```

Procedure JuliaSet(p, q : Real; LocX, LocY, ResX, ResY, Color :
                  Integer);

```

```

  Begin
    SphrPix := False;
    LocX := LocX + MaxX Div 2 - ResX;
    LocY := LocY + MaxY Div 2 - ResY;
    Scan(p, q, -1.4, -1.4, 1.4, 1.4, LocX, LocY, 0, ResX, ResY,
        Color)
  End;

```

```

Procedure MapJuliaSetToSphere(p, q : Real; LocX, LocY, Rad, ResX,
                              ResY, Color : Integer);

```

```

  Begin

```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
SphrPix := True;
InitPlotting(180, 0);
InitPerspective(False, 0, 0, 0, 0);
Scan(p, q, -2.2, -2.2, 2.2, 2.2, LocX, LocY, Rad, ResX,
                                           ResY, Color)
End;
```

```
{
```

Blended Sky

```
}
Sky - blend a sky color from the horizon to the zenith
```

```
Procedure Sky(LocX, LocY, ResX, ResY : Integer; Color : Byte);
```

```
Procedure CalcSky(Dir : TDA; Var Inten : Byte);
```

```
Const
```

```
Small      = 1E-03;
```

```
HorInten = 0.25;
```

```
ZenInten = 1.00;
```

```
Var
```

```
sin2, cos2 : Real;
```

```
x2, y2, z2 : Real;
```

```
Begin
```

```
x2 := Sqr(Dir[0]);
```

```
y2 := Sqr(Dir[1]);
```

```
z2 := Sqr(Dir[2]);
```

```
If (z2 = 0) Then z2 := Small;
```

```
sin2 := z2 / (x2 + y2 + z2);
```

```
cos2 := 1.0 - sin2;
```

```
Inten := Round( (cos2 * HorInten + sin2 * ZenInten) *
                                                         MaxInten )
```

```
End;
```

```
Var
```


THE Z-BUFFER RENDERING PROGRAM

```

i, j      : Integer;
Dir       : TDA;
Tmp, Eye  : TDIA;
HalfResX  : Integer;
HalfResY  : Integer;
EyeDist   : Integer;
OffX      : Integer;
OffY      : Integer;
Inten     : Byte;

Begin
  HalfResX := ResX Div 2;
  HalfResY := ResY Div 2;
  OffX := 0;
  OffY := -HalfResY - HalfResY Div 2 - HalfResY Div 4;
  EyeDist := -HalfResY;
  VecInt(OffX, OffY, EyeDist, Eye);
  For i := 0 To ResX-1 Do
    For j := 0 To ResY-1 Do
      Begin
        VecInt(i-HalfResX, j-HalfResY, 0, Tmp);
        VecSubInt(Tmp, Eye, Dir);
        VecNormalize(Dir);
        CalcSky(Dir, Inten);
        If HalfScale Then
          PutPixel( LocX + i, LocY + j + HalfResY ,
                    Color, Inten )
        Else
          Begin
            PutPixel( LocX + i*2 , LocY + j*2 , Color,
                      Inten );
            PutPixel( LocX + i*2+1, LocY + j*2 , Color,
                      Inten );
            PutPixel( LocX + i*2 , LocY + j*2+1, Color,
                      Inten );
            PutPixel( LocX + i*2+1, LocY + j*2+1, Color,
                      Inten )
          End
        End
      End
    End
  End;

```

Figure 13-3. Listing of *AddOns.inc* File

If the parameter *SphrPix* is true, the Julia set is projected onto the surface of a sphere. The code for this is the same as for plotting pixels to the screen for the ordinary Julia set except instead of plotting each pixel, its information is passed in a call to the procedure *PlotSphrPix*. The coordinates for the lower left corner of the display area, the coordinates of the current point in the display, the radius of a sphere, and the pixel color and intensity pass to this procedure. It multiplies the point coordinates by two and calls *PutDitheredSphere* to paint four pixels to the screen (the designated point and three surrounding it in a square). The *PutDitheredSphere* procedure converts the coordinates of the point to angles on the sphere and determines the *x*, *y*, and *z* coordinates of the point on the sphere. It then calls *MapCoordinates* to convert this three-dimensional information to the appropriate pixel location on the two-dimensional screen. The pixel is then painted to the screen with *PutPixel*. This completes the *PutDitheredSphere*, *PlotSphrPix*, and *Scan* procedures.

c = p + iq ACP = attractive cycle period		
p	q	Description
0.31	0.04	
-0.11	0.6557	JSet ACP=3 before decay into Cantor set after decay into Cantor set
-0.194	0.6557	
-0.12	0.74	
0.0	1.0	lightning dendrite
-0.74543	0.11301	seahorse valley c-value
-1.25	0.0	parabolic case c>1.25 ACP=2, c<1.25 ACP=4
-0.481762	-0.531657	
-0.39054	-0.58679	
-0.15652	-1.03225	dendrite with beads (secondary MSet)
0.11031	-0.67037	Fatou dust
0.27334	0.00742	parabolic case small c ACP=20

Figure 13-4. Types of Julia Set

THE Z-BUFFER RENDERING PROGRAM

Now let's look at the Julia set. This procedure is passed the parameters p and q (parameters for the Julia set), $LocX$ and $LocY$, (center of the display area), $ResX$ and $ResY$, (size of the display in pixels), and $Color$, (display color). The procedure sets $SphrPix$ to false so the Julia set will not be mapped onto a sphere. It then changes $LocX$ and $LocY$ to the bottom left corner of the display area. It then calls *Scan* to generate the Julia set display.

The procedure *MapJuliaSetToSphere* is the same as this except it first sets $SphrPix$ to true causing the Julia set to be mapped to a sphere. It then calls *InitPlotting* and *InitPerspective* (described in Chapter 2) to convert the three-dimensional information to a two-dimensional display.

Creating a Blended Sky

Another specialized procedure is *Sky*, which creates a sky color blended from horizon to zenith. This procedure creates coordinates for the eye position of a viewer and, for each position on the screen, creates a vector from the eye position to that position and then normalizes this vector. It then calls *CalcSky*, which finds the squares of the cosine and sine of the angle of this normalized vector to the xy plane. The sky intensity at that point is the horizon intensity (0.25) multiplied by the cosine squared of the angle plus the zenith intensity (1.0) multiplied by the sine squared of the angle. This completes *CalcSky*. The *Sky* procedure then paints the proper pixels on the screen with the designated color and just-computed intensity.

Creating and Working with Z-Buffer Databases

You now can produce a scene, given a database containing an array consisting of a grid of equally-spaced points in the x - y plane and the height associated with each. But thus far, nothing has been said about how to create such arrays. In this chapter, let's look at how to generate, examine, and modify databases for such shapes as hemispheres and hemitoroids. Let's also examine the solution of various equations and the plotting of magnetic fields. In the next chapter you will generate data bases for scenes containing fractals of mountains, Julia and Mandelbrot sets, quaternions, dragons, and solids of revolution.

Generating the Database for a Hemisphere

Figure 14-1 is a listing of the program *HmSphere.Pas*, which generates the database for painting a hemisphere on the screen using z-buffering. The program clears the screen and displays, "Creating Hemisphere Height Buffer". It then calls *ClearHeightBuffer*. This procedure, described in Chapter 11, clears the array of all height data that is temporarily stored. Next, the program sets the parameter *Res* (the picture resolution) to the value of *MaxRes* (160). The object data file is then named *HmSphere*. The program then calls the procedure *CreateHemisphereHeightBuffer*. This procedure determines *Offset*, (where the location of the object will be in the x - y plane). This is exactly in the center of the plane. The radius for the hemisphere, R , is now set to one less than the distance from the center to the edge of the plane. Next, the procedure calls *HeightBufferScalingFactor*, described in Chapter 11, to determine the proper scaling factor for the height values. The procedure then enters a *For* loop, which iterates for every integer angle Φ from 0 to 89. This is the number of

passes required to move around a quarter of the hemisphere in one-degree steps. Because the hemisphere is symmetrical, at each step four points can be computed to cover the four quarters, thus completing data for the hemisphere. Within the loop, the procedure computes the radius times the cosine of the angle and the radius of the sine of the angle for each of four angles, 90 degrees apart. (These values are computed separately, but actually the sines and cosines of the last three angles are simply related to the first angle, so you might want to achieve a small speed-up to compute the last three angles by simple trigonometric relations, instead of recalculating.) The two values just computed are actually the x and y coordinates of the points on a circle that forms the base of the hemisphere. The procedure now enters a second, nested *For* loop, which iterates for integer steps of the height angle θ from 0 to 90 degrees. At each step, the x , y , and z coordinates of a point on the surface of the hemisphere are computed for each of the four quarters of the hemisphere. These are all converted to integers, and at the location (x,y) in the *Height* array, the value of the height z is stored. After the loops have finished iterating, the procedure returns to the main program, which then calls *SaveHeightBuffer* (described in Chapter 11) to save the data to a disk file called *HmSphere.dat*.

```
{
  

Generate Hemisphere Database


}
```

Uses

Crt;

{ \$I Math.Inc }

{ \$I Render.Inc }

Procedure CreateHemisphereHeightBuffer;

CREATING AND WORKING WITH Z-BUFFER DATABASES

Var

```
X, Y, R, Offset, Z      : Integer;
PhiR, ThetaR            : Integer;
Phi, Theta              : Real;
RSinPhi1, RCosPhi1      : Real;
RSinPhi2, RCosPhi2      : Real;
RSinPhi3, RCosPhi3      : Real;
RSinPhi4, RCosPhi4      : Real;
SinTheta, CosTheta      : Real;
```

Begin

```
Offset := Res Div 2;
R := Offset - 1;
MaxHeight := R;
HeightBufferScalingFactor;
For PhiR := 0 to 89 Do { take advantage of symmetry }
```

Begin

```
Phi := Radians( PhiR );
RCosPhi1 := R * Cos( Phi );
RSinPhi1 := R * Sin( Phi );
Phi := Radians( PhiR + 90 );
RCosPhi2 := R * Cos( Phi );
RSinPhi2 := R * Sin( Phi );
Phi := Radians( PhiR + 180 );
RCosPhi3 := R * Cos( Phi );
RSinPhi3 := R * Sin( Phi );
Phi := Radians( PhiR + 270 );
RCosPhi4 := R * Cos( Phi );
RSinPhi4 := R * Sin( Phi );
For ThetaR := 0 To 90 Do
```

Begin

```
Theta := Radians( ThetaR );
CosTheta := Cos( Theta );
SinTheta := Sin( Theta );
Z := Round( CosTheta * Scaling ) Div 2;
X := Round( SinTheta * RCosPhi1 ) + Offset;
Y := Round( SinTheta * RSinPhi1 ) + Offset;
Height[X,Y] := Z;
X := Round( SinTheta * RCosPhi2 ) + Offset;
Y := Round( SinTheta * RSinPhi2 ) + Offset;
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```

        Height[X,Y] := Z;
        X := Round( SinTheta * RCosPhi3 ) + Offset;
        Y := Round( SinTheta * RSinPhi3 ) + Offset;
        Height[X,Y] := Z;
        X := Round( SinTheta * RCosPhi4 ) + Offset;
        Y := Round( SinTheta * RSinPhi4 ) + Offset;
        Height[X,Y] := Z
    End
End
End;

{
  Main Program
}

Begin
  ClrScr;
  WriteLn('Creating Hemisphere Height Buffer');

  ClearHeightBuffer;

  Res := MaxRes;

  ObjectFile := 'HmSphere';

  CreateHemisphereHeightBuffer;

  SaveHeightBuffer(ObjectFile)
End.
```

Figure 14-1. Listing of Program to Generate a Hemisphere.

Generating the Database for a Hemitoroid

The main program to create the database for a hemitoroid is listed in Figure 14-2. It is the same as the main program for a hemisphere, except for the message it displays on the screen, and the fact that it calls the procedure *CreateHemiToroidHeightBuffer* instead of *CreateHemiSphereHeightBuffer*. This procedure is passed two parameters, *Rho* and *R*. The parameter *Rho* (0.75) is multiplied by half the display resolution to obtain a new *Rho* which is the radius of the doughnut; the parameter *R* (0.25) is multiplied by half the display resolution to obtain a new *R*, which is the radius of the doughnut cross-section. The procedure begins an outer *For* loop like the procedure for the sphere, iterating for angle *ThetaR* from 0 to 89, one quarter of the circle formed by the half-doughnut on the base plane. At each iteration the coordinates of four points on this circle are computed. Then an inner *For* loop iterates for angle *PhiR* from 0 to 180. To create the hemisphere data, the inner loop swept through a quarter circle in height at a constant radius from the center of base circle. For the hemitoroid, the inner loop sweeps through a half circle at a constant radius from the point on the circumference of the base circle. At each step, the *x*, *y*, and *z* coordinates of a point on the surface of the hemitoroid are computed for each of the four quarters of the hemitoroid. These are all converted to integers, and at the location (*x*,*y*) in the *Height* array, the value of height *z* is stored. After the loops have finished iterating, the procedure returns to the main program, which then calls *SaveHeightBuffer* (described in Chapter 11) to save the data to a disk file called *HmToroid.dat*.

Note at the beginning of this file there are two sets of given constants. First, the parameter *Zscale* is set to 1.0 and the name of the object data file, *ObjF* is set to *HmToroid*. This produces the database file just described. Following is another set of values for the same constants, in which the parameter *Zscale* is set to 2.0 and the name of the object data file, *ObjF* is set to *Bowl*. This set of constants has been commented out. If you remove the curly brackets from this set of constants and use them to comment out the first set of values, you'll create the data base for a bowl-shaped object.

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
{
```

```
    Generate HemiToroid Database
```

```
}
```

```
Uses
```

```
    Crt;
```

```
    {$I Math.Inc}
```

```
    {$I Render.Inc}
```

```
Const
```

```
    ZScale = 1.0;
```

```
    ObjF = 'HmToroid';
```

```
{
```

```
Const
```

```
    ZScale = 2.0;
```

```
    ObjF = 'Bowl';
```

```
}
```

```
Procedure CreateHemiToroidHeightBuffer(Rho, R : Real);
```

```
Var
```

```
    ThetaR, PhiR : Integer;
```

```
    X, Y, Z, Offset : Integer;
```

```
    Theta, Phi : Real;
```

```
    CosTheta1, SinTheta1 : Real;
```

```
    RhoCosTheta1, RhoSinTheta1 : Real;
```

```
    CosTheta2, SinTheta2 : Real;
```

```
    RhoCosTheta2, RhoSinTheta2 : Real;
```


CREATING AND WORKING WITH Z-BUFFER DATABASES

```
CosTheta3, SinTheta3      : Real;
RhoCosTheta3, RhoSinTheta3 : Real;
CosTheta4, SinTheta4      : Real;
RhoCosTheta4, RhoSinTheta4 : Real;
RCosPhi, RSinPhi          : Real;
Zfactor                   : Real;

Begin
  Rho := Rho * Res / 2;
  R := R * Res / 2;
  Offset := Res Div 2;
  MaxHeight := Round(R * 2);
  HeightBufferScalingFactor;
  Zfactor := ZScale * MaxHeight / 4;
  For ThetaR := 0 To 89 Do
    Begin
      Theta := Radians( ThetaR );
      CosTheta1 := Cos( Theta );
      SinTheta1 := Sin( Theta );
      RhoCosTheta1 := Rho * CosTheta1;
      RhoSinTheta1 := Rho * SinTheta1;

      Theta := Radians( ThetaR + 90 );
      CosTheta2 := Cos( Theta );
      SinTheta2 := Sin( Theta );
      RhoCosTheta2 := Rho * CosTheta2;
      RhoSinTheta2 := Rho * SinTheta2;
      Theta := Radians( ThetaR + 180 );
      CosTheta3 := Cos( Theta );
      SinTheta3 := Sin( Theta );
      RhoCosTheta3 := Rho * CosTheta3;
      RhoSinTheta3 := Rho * SinTheta3;

      Theta := Radians( ThetaR + 270 );
      CosTheta4 := Cos( Theta );
      SinTheta4 := Sin( Theta );
      RhoCosTheta4 := Rho * CosTheta4;
      RhoSinTheta4 := Rho * SinTheta4;

      For PhiR := 0 To 180 Do
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
Begin
  Phi := Radians( PhiR );
  RCosPhi := R * Cos( Phi );
  RSinPhi := R * Sin( Phi );
  Z := Round(RSinPhi * Zfactor);
  X := Round(RCosPhi * CosTheta1 + RhoCosTheta1)
      + Offset;
  Y := Round(RCosPhi * SinTheta1 + RhoSinTheta1)
      + Offset;

  Height[X,Y] := Z;
  X := Round(RCosPhi * CosTheta2 + RhoCosTheta2)
      + Offset;
  Y := Round(RCosPhi * SinTheta2 + RhoSinTheta2)
      + Offset;

  Height[X,Y] := Z;
  X := Round(RCosPhi * CosTheta3 + RhoCosTheta3)
      + Offset;
  Y := Round(RCosPhi * SinTheta3 + RhoSinTheta3)
      + Offset;

  Height[X,Y] := Z;
  X := Round(RCosPhi * CosTheta4 + RhoCosTheta4)
      + Offset;
  Y := Round(RCosPhi * SinTheta4 + RhoSinTheta4)
      + Offset;

  Height[X,Y] := Z
End
End
End;

{
  Main Program
}
```

```
Begin
  ClrScr;
  WriteLn('Creating HemiToroid Height Buffer');
```


CREATING AND WORKING WITH Z-BUFFER DATABASES

```
ClearHeightBuffer;  
  
Res := MaxRes;  
  
ObjectFile := ObjF;  
  
CreateHemiToroidHeightBuffer(0.75, 0.25);  
  
SaveHeightBuffer(ObjectFile)  
End.
```

Figure 14-2. Listing of Program to Generate a Bowl or Hemitoroid.

Generating the Database for Equations

The same program is used to generate database files for four different equations. The file for this program is *PlotEqn.Pas*. It is listed in Figure 14-3. At the beginning of the file listing, you will see a section for each of the four equations, containing the name of the object data file for the particular equation, the span of the equation (which determines the magnification needed for the equation to fill the screen) and the necessary function to plot the values of z , given a particular set of values for x and y . The equations are the same as those used in the solid modeling program. They are:

$$z = \frac{75}{x^2 + y^2 + 1} \quad (\text{Equation 14-1})$$

$$z = 6 (\cos(xy) + 1) \quad (\text{Equation 14-2})$$

$$z = 20 (\sin(\sqrt{x^2 + y^2}) + 1) \quad (\text{Equation 14-3})$$

$$z = 10(1 + \sqrt{x^2 + y^2} + \sin(0.1(x^2 + y^2)^2) + \sin(xy)) \quad (\text{Equation 14-4})$$

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
{
```

```
    Generate Equation Plot Database
```

```
}
```

```
Uses
```

```
    Dos, Crt;
```

```
{$I Math.Inc}
```

```
{$I Graph.Inc}
```

```
{$I Render.Inc}
```

```
{
```

```
    Equations
```

```
}
```

```
{
```

```
Const
```

```
    ObjF = 'PlotEqn1';
```

```
    Span = 5;
```

```
Function zf(x, y : Real) : Real;
```

```
Var
```

```
    c : Real;
```

```
Begin
```

```
    c := Sqr(x) + Sqr(y);
```

```
    zf := 75 / ( c + 1 )
```

```
End;
```

```
}
```


CREATING AND WORKING WITH Z-BUFFER DATABASES

```
{  
  Const  
    ObjF = 'PlotEqn2';  
    Span = 5;
```

```
Function zf(x, y : Real) : Real;
```

```
  Begin  
    zf := 6 * (Cos(x * y) + 1);  
  End;
```

```
}
```

```
{  
  Const  
    ObjF = 'PlotEqn3';  
    Span = 5;
```

```
Function zf(x, y : Real) : Real;
```

```
  Var  
    c : Real;
```

```
  Begin  
    c := Sqr(x) + Sqr(y);  
    zf := 20 * ( Sin( Sqrt(c) ) + 1 )  
  End;
```

```
}
```

```
{  
  Const  
    ObjF = 'PlotEqn4';  
    Span = 2.75;
```

```
Function zf(x, y : Real) : Real;
```

```
  Var  
    c : Real;
```

```
  Begin  
    c := Sqr(x) + Sqr(y);
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
    zf := 10 * ( 1 + Sqrt(c) + Sin( Sqr(c)*0.1 ) + Sin(x*y) );
End;
}

{


---


}
```

```
Procedure CreateEquationPlotHeightBuffer(Xlft, Xrgt, Ybot, Ytop :
Real);
```

```
Var
```

```
    ix, iy, iz : Integer;
    x, y       : Real;
    dx, dy     : Real;
```

```
Begin
```

```
    MaxHeight := Res;
    HeightBufferScalingFactor;
```

```
    dx := (Xrgt - Xlft) / (Res-1);
    dy := (Ytop - Ybot) / (Res-1);
```

```
    For ix := 0 To Res-1 Do
```

```
        Begin
```

```
            x := Xlft + ix * dx;
```

```
            For iy := 0 To Res-1 Do
```

```
                Begin
```

```
                    y := Ybot + iy * dy;
```

```
                    iz := Round(zf(x,y));
```

```
                    If (iz < 0) Then
```

```
                        Begin
```

```
                            ExitGraphics;
```

```
                            WriteLn('Adjust the equation : z value  
                                less than zero');
```

```
                            Delay(2000);
```

```
                            Halt
```

```
                        End;
```


CREATING AND WORKING WITH Z-BUFFER DATABASES

```
        Height[ix,iy] := iz;
        CartesianPlot3D(ix - Res Div 2, iy - Res Div 2,
                        iz, (iz * MaxCol) Mod 255);
    End
End
End;

{
    Main Program
}

Begin
    ClrScr;
    WriteLn('Creating Equation Plot Height Buffer');
    WriteLn;
    ClearHeightBuffer;

    Res := MaxRes;

    ObjectFile := ObjF;

    InitPerspective(False,0,0,500,500);
    InitPlotting(240,18);

    InitGraphics;
    { Xlft, Xrgt, Ybot, Ytop }
    CreateEquationPlotHeightBuffer(-Span, Span, -Span, Span);

    SaveHeightBuffer(ObjectFile);

    ExitGraphics
End.
```

Figure 14-3. Listing of Program to Generate Plots of Equations

The above information for each of the four equations is commented out. To run the program, select one set of equation information and remove the comments surrounding it (the curly brackets) so that the program can generate that database. The program begins by clearing the screen and displaying, "Creating Equation Plot Height Buffer". The procedure *ClearHeightBuffer* is then called to clear the height buffer and resolution of the display is set to 160. The name of the object data file is set to that contained in *ObjF*. The program then calls *InitPerspective*, *InitPlotting*, and *InitGraphics* to prepare for generating a graphics display. (These were not needed in the previous programs, because the information was not displayed; it was only transferred to the datafile. In this program, you'll display the data as it is created, so you can correct any mistakes and try again until you are satisfied that the right information is going to the file.) Next, the program calls a procedure *CreateEquationPlotHeightBuffer*. This procedure clears the height buffer and determines the height scaling factor. It then sets up the increments for increasing adjacent values of x and y , enters a pair of nested *For* loops, which together iterate through every member of the height buffer array. At each iteration, the values of x and y are determined. The appropriate equation function is then used to determine the height (z), rounded off to an integer. If the height is less than zero, the procedure returns to the text mode, displays, "Adjust the equation: z value less than zero", delays for two seconds, and halts the computer. You have to restart and change the equation parameters in your editor before recompiling the program. If the height value is not less than zero, it is stored in the proper member of the height array and projected to two dimensions and plotted on the screen with the procedure *CartesianPlot3D*. The color of the plotted pixel is a function of the height, modulo 255, not to exceed the color range of the VGA mode being used. These colors bear no relation to the lighting of the object (which requires running of *Render.Pas*) but they do show the figure shape for study and reference. Once all of the members of the height buffer array have been filled, the procedure returns to the main program, where the procedure *SaveHeightBuffer* saves the array to the specified object data file.

Generating a Magnetic Field Database

The program *MagnFlds.Pas* generates the database for the magnetic fields produced by two parallel wires, one having a current of 100 amperes in one direction and the other having a current of 40 amperes in the opposite direction. The main program clears the screen and displays, "Creating Magnetic Field Height Buffer" and calls *ClearHeightBuffer* to clear the height buffer array. The display resolution is set to maximum (160) and the object data file named *MagnFlds*. Then the procedure *Parameters* is called. This procedure simply sets the values of the parameters in the magnetic field equation. *InitGraphics* enters the graphics mode. (This is for creating a simple two-dimensional display of the field.) Next, the program calls the procedure *CreateMagneticField*. This procedure sets the maximum height and the scaling factor, enters a pair of nested *For* loops which iterate once for every combination of x and y in the height array. At each point, the procedure computes the strength of the magnetic field and stores it as height in the height array. It then plots a pixel to the screen at the x - y location, using the height to designate the color and intensity information. After all members of the array are computed, stored, and plotted, the procedure returns to the main program, which calls *SaveHeightBuffer* to store the height array data in a database file. It then leaves the graphics mode and terminates.

Finding the Highest and Lowest Points in the Height Array

It may be that after you create a datafile of height data and use the *Render.Pas* program to display it, you find that either the variation in height over the display is too small, so that the display is uninteresting, or else the variation is too great and the program cannot handle it. Rather than try to guess how parameters need to be modified to bring the height data into the proper range, it is useful to have a program which will extract the highest and lowest values from the object database file. The program *FindLmts.Pas*, listed in Figure 14-5, performs this task.

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
{
```

```
Generate Magnetic Fields Database
```

```
}
```

```
Uses
```

```
Dos, Crt;
```

```
{$I Math.Inc}
```

```
{$I Graph.Inc}
```

```
{$I Render.Inc}
```

```
Var
```

```
Xa, Ya, Xb, Yb : Integer;
```

```
Ia, Ib          : Integer;
```

```
k              : Real;
```

```
Procedure Parameters;
```

```
Begin
```

```
  Xa := Round(Res * 0.34); { electric wire locations }
```

```
  Ya := Round(Res * 0.60);
```

```
  Xb := Round(Res * 0.68);
```

```
  Yb := Round(Res * 0.40);
```

```
  Ia := 100; { electrical current levels }
```

```
  Ib := -40;
```

```
  k := 2E-07 { physical constants }
```

```
End;
```

```
Procedure CreateMagneticFieldHeightBuffer;
```


CREATING AND WORKING WITH Z-BUFFER DATABASES

```
Var
  X, Y      : Integer;
  Da, Db    : Real;
  Hgt, B    : Real;
  Inten     : Byte;
  Z         : Integer;

Begin
  MaxHeight := Res Div 2 - 1;
  HeightBufferScalingFactor;

  For X := 0 To Res Do
    For Y := 0 To Res Do
      Begin
        Da := Sqrt(Sqr(Xa - X) + Sqr(Ya - Y));
        Db := Sqrt(Sqr(Xb - X) + Sqr(Yb - Y));
        If Not(Da = 0) And Not(Db = 0) Then
          B := k * ( Ia / Da + Ib / Db )
        Else
          B := 0;
        Hgt := Abs(B * 1E+08 + 0.5);

        Z := Round(Hgt * 0.25);
        Height[X,Y] := Z;

        Inten := Round(Hgt) And $FF;
        PutPixel(X ,199-Y, Inten Div MaxCol, Inten Mod
                                                    MaxInten)
      End
    End
  End;
```

```
{

```

Main Program

```
}
Begin
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
ClrScr;
WriteLn('Creating Magnetic Field Height Buffer');

ClearHeightBuffer;

Res := MaxRes;

ObjectFile := 'MagnFlds';

Parameters;

InitGraphics;

CreateMagneticFieldHeightBuffer;

SaveHeightBuffer(ObjectFile);

ExitGraphics
End.
```

Figure 14-5. Listing of Program to Generate Magnetic Fields Data

```
{
    Program to find highest and lowest z-values
}

Uses
    Crt;

{$I Render.Inc}

Var
    I, J : Integer;
    A, B, C : Integer;
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
x1, y1   : Integer;
x2, y2   : Integer;

Begin
  ClrScr;
  WriteLn('Find Locations of Highest and Lowest Points');
  WriteLn;

  GetObjectFile;

  ClearHeightBuffer;

  LoadHeightBuffer(ObjectFile);

  B := 0;
  C := 32767;
  For I := 0 To Res Do
    For J := 0 To Res Do
      Begin
        A := Height[I,J];
        If (A > B) Then
          Begin
            x1 := I;
            y1 := J;
            B := A;
          End;
        If (A < C) Then
          Begin
            x2 := I;
            y2 := J;
            C := A;
          End;
        End;
      End;
    End;
  WriteLn;
  WriteLn('High Point is (' ,x1,',',',y1,') = ',B);
  WriteLn('Low   Point is (' ,x2,',',',y2,') = ',C);
  Repeat Until Keypressed
End.
```

Figure 14-6. Listing of Program to Find Highest and Lowest Values in Array

The program first clears the screen and then displays, "Find Locations of Highest and Lowest Points". Next, it calls *GetObjectFile*, which asks you, "Enter File Name =>". You need to enter an object database file name without the extension. (The program will automatically add the extension *.DAT*.) The program then calls *ClearHeightBuffer*, which clears the height buffer array. Next, it calls *LoadHeightBuffer*, which displays, "Loading nnnn.Dat" (where *nnnn* is the name of the database file that you specified) and loads the file into the height array. The program then enters a pair of nested *For* loops which together scan every member of the height array. At each loop iteration, the current height value is compared with the contents of *B* (which begins at 0) and *C* (which begins at 32767). If the current value is larger than *B*, it replaces *B* and its coordinates are saved; if it is smaller than *C*, it replaces *C* and its coordinates are saved. When the loops are finished, *B* contains the highest value in the array and *C* contains the lowest. The program then displays:

```
High Point is (x1,y1) = B
Low Point is (x2,y2) = C
```

where *B* is actually the value of the high point, (*x1,y1*) are actually the values of its coordinates, *C* is actually the value of the low point, and (*x2,y2*) are actually the values of its coordinates. The program continues to loop, displaying this data, until a key is pressed.

Viewing a Slice of the Database

In analyzing a three-dimensional function to be displayed with the *z*-buffering technique, it is frequently useful to view a cross-section of the displayed data. The program *ViewSlic.Pas*, listed in Figure 14-7, will allow you to specify a particular value for the *x* coordinate and draw a picture of the value of *z* for every *y*. Or it will let you specify a particular value for the *y* coordinate and draw a picture of the value of *z* for every *x*. The program begins by clearing the screen and then displaying, "Display Slice of *z*-buffer data". Next, it calls *GetObjectFile*, which asks you, "Enter File Name =>". You need to enter an object database file name without the extension. (The program will automatically add the extension *.DAT*.) The program then calls

ClearHeightBuffer, which clears the height buffer array. Next, it calls *LoadHeightBuffer*, which displays, "Loading nnnn.Dat" (where *nnnn* is the name of the database file you specified) and then loads the file into the height array. The program then enters a *Repeat* loop which iterates as long as the value of *Sl* is either 0 or 1. The program displays, "Xslice=0 or Yslice=1 =>". If you want a display of the *xz* plane for a particular value of *y*, you enter a 0. If you want a display of the *yz* plane for a particular value of *x*, you enter a 1. If you want to quit the program, you enter anything else. The entered character becomes the value of *Sl*. Next, if the value of *Sl* is one, the program displays, "Choose a slice in X =>" and you enter a number for the value of *x*. The program then starts the graphics mode and calls the procedure *TestHeightBufferYZPlane* to compute and display the slice profile. The program enters a waiting loop until a key is pressed. If the value of *Sl* is zero, the program displays, "Choose a slice in Y =>". You then enter a number for the value of *y* at which the slice is to be taken. The program then starts the graphics mode and calls the procedure *TestHeightBufferXZPlane* to compute and display the slice profile. Then the program enters a waiting loop until a key is pressed. After either of these actions takes place (or neither, depending upon which key was entered to set the value of *Sl*) the program calls *ExitGraphics* to leave the graphics mode and return to the text mode. If *Sl* was zero or one, the program returns to the beginning of the *Repeat* loop, providing the opportunity to select another slice for viewing. If *Sl* has any other value, the program terminates.

```
{
{
}
```

Program to display slices of the database

```
Uses
  Dos, Crt;
```

```
{ $I Math.Inc }
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
{ $I Graph.Inc }
```

```
{ $I Render.Inc }
```

```
Var
```

```
    Max : Integer;
```

```
Procedure TestHeightBufferXZPlane(Slice : Integer);
```

```
    Var
```

```
        I, J : Integer;
```

```
    Begin
```

```
        J := Slice;
```

```
        For I := 0 To Res Do
```

```
            PutPixel(I, Round(MaxY - 100 * Height[I,J] / Max),  
                    MaxCol, MaxInten)
```

```
    End;
```

```
Procedure TestHeightBufferYZPlane(Slice : Integer);
```

```
    Var
```

```
        I,J : Integer;
```

```
    Begin
```

```
        I := Slice;
```

```
        For J := 0 To Res Do
```

```
            PutPixel(J, Round(MaxY - 100 * Height[I,J] / Max),  
                    MaxCol, MaxInten)
```

```
    End;
```

```
Procedure FindMax;
```

```
    Var
```

```
        i, j : Integer;
```

```
    Begin
```


CREATING AND WORKING WITH Z-BUFFER DATABASES

```
Max := 0;
For i := 0 To MaxRes Do
  For j := 0 To MaxRes Do
    If (Height[i,j] > Max) Then Max := Height[i,j]
                                { find max height }
End;
```

```
{
```

Main Program

```
}
```

```
Var
```

```
  Slice : Integer;
```

```
  Sl : Byte;
```

```
Begin
```

```
  ClrScr;
```

```
  WriteLn;
```

```
  WriteLn('Display Slice of z-Buffer Data');
```

```
  GetObjectFile;
```

```
  ClearHeightBuffer;
```

```
  LoadHeightBuffer(ObjectFile);
```

```
  FindMax;
```

```
  Repeat
```

```
    Write('Xslice=0 or Yslice=1 => ');
```

```
    ReadLn(Sl);
```

```
    If Sl = 1 then
```

```
      Begin
```

```
        Write('Choose a slice in x => ');
```

```
        ReadLn(Slice);
```

```
        InitGraphics;
```

```
        TestHeightBufferYZPlane(Slice)
```

```
        Repeat Until Keypressed;
```



```

End;
If S1 = 0 Then
Begin
    Write('Choose a slice in y => ');
    ReadLn(Slice);
    InitGraphics;
    TestHeightBufferXZPlane(Slice);
    Repeat Until Keypressed;
End;
ExitGraphics;
Until (S1 <> 0) and (s1 <> 1);
End.

```

Figure 14-7. Program to View Slice of Height Buffer

The procedure *TestHeightBufferXZPlane* takes the heights for all y values in the height buffer, at the value of x taken as the slice point, scales, and plots them to the screen in bright white. The scaling is done over a range of 100 pixels, by dividing the height values by the maximum height value and multiplying the result by 100. The same process occurs for the procedure *TestHeightBufferYZPlane*, except that the procedure takes the heights for all x values in the height buffer at the value of y taken as the slice point, and scales, and plots them to the screen in bright white.

Removing Discontinuous Zeroes

Sometimes the height array database contains erroneous, isolated points recorded at zero height. These points are not part of a realistic definition of the surface and therefore need to be corrected before the database is sent to the rendering program. The program *Smooth.Pas*, listed in Figure 14-8, performs this operation. The program first clears the display screen and then displays, “z-Buffer Data Correction Program”. Next, it calls *GetObjectFile*, which asks you, “Enter File Name =>”. You then need to enter an object database file name without the extension. (The program will automatically add the extension *.DAT*.) The program then calls *ClearHeightBuffer*, which clears the height buffer array and calls *LoadHeightBuffer*, which displays, “Loading nnnn.Dat” (where *nnnn* is the name of the database file that you specified) to load the file into the height array. Next, it displays the line, “Correcting Height Buffer”. It enters a pair of nested *For* loops which iterate once

CREATING AND WORKING WITH Z-BUFFER DATABASES

for every member of the height array. If a member of the array is found to contain zero height and neither of the points in the columns on either side of it is zero, the height is set to the average of the values of these two adjacent points. The entire procedure repeats again, except that each point is checked against the points in the rows on either side. Finally, the program calls *SaveHeightBuffer* to save the corrected height array as the new database file.

```
{
```

Program to remove discontinuous zeroes from the data
--

```
}
```

```
Uses
```

```
  Crt;
```

```
{$I Render.Inc}
```

```
Var
```

```
  I, J, A, B, C : Integer;
```

```
Begin
```

```
  ClrScr;
```

```
  WriteLn('z-Buffer Data Correction Program');
```

```
  WriteLn;
```

```
  GetObjectFile;
```

```
  ClearHeightBuffer;
```

```
  LoadHeightBuffer(ObjectFile);
```

```
  WriteLn;
```

```
  WriteLn('Correcting Height Buffer');
```


CREATING AND WORKING WITH Z-BUFFER DATABASES

```
For J := 0 To Res Do
  For I := 1 To Res-1 Do
    Begin
      A := Height[I,J];
      B := Height[I-1,J];
      C := Height[I+1,J];
      If (A=0) And Not(B=0) And Not(C=0) Then
        Height[I,J] := Round((B + C) * 0.5)
      End;
    End;

  For I := 0 To Res Do
    For J := 1 To Res-1 Do
      Begin
        A := Height[I,J];
        B := Height[I,J-1];
        C := Height[I,J+1];
        If (A=0) And Not(B=0) And Not(C=0) Then
          Height[I,J] := Round((B + C) * 0.5)
        End;
      End;
    End;

  SaveHeightBuffer(ObjectFile)
End.
```


Fractal Programs to Generate Databases

In this chapter you'll look at methods for using fractals to generate databases for rendering by the z-buffering method. Some fractals produce pictures similar to things found in nature. You will generate several fractal mountains and convert the traditional Mandelbrot and Julia sets into three-dimensional representations, resulting in unusual and hauntingly familiar scenes. Finally, you'll create three-dimensional slices of quaternion Julia Sets yielding dragons and solids of revolution.

Generating Fractal Mountains

The program and procedures in the file *Mountain.Pas*, listed in Figure 15-1, generate fractal mountain databases. The program begins by displaying:

Mountain Scene Generator
By Chris Watkins

Next the procedure *ClearHeightBuffer* zeroes the array of height data. The program then calls the procedure *Initialization*. This procedure determines the value of *Points*, which is 2 raised to the *NumberLevels* (7) power. It determines the value of *Space*, which is 2 raised to the *MaximumLevel* - *NumberLevels* (0) power. It then sets *MaxHeight* to the value of *Roughness* (100) and calls the procedure *HeightBufferScalingFactor* to determine the proper scaling factor for height data. The procedure then returns to the main program which calls the procedure *CalculateFractalHeights* to calculate all of the data to fill the height array. This procedure starts with a *For* loop that iterates once for each level from one up to the

specified number of levels. At each level, the loop sets the values for *Deviation*, *HalfDeviation*, *Skip*, and *Offset*. The procedure then displays "Pass Level," followed by the level number. The procedure then interpolates in *x* by means of a pair of nested *Repeat* loops. What happens here, is that at the first level, a point is selected at the center of the top line of the picture and its height becomes the average of the heights of the points at the two ends of the line, plus a random height factor. The same process is then applied to the point at the center of the bottom line of the array. The procedure interpolates in *y*, using the same process, but determining the height for points midway in the first and last columns, using the average height of the points at the ends of the column plus a random factor. Finally the same process determines the height of a point in the center of the display, using the average of the heights of points on the opposite ends of a diagonal through the center, plus a random factor. At the next level of the *For* loop, the spacings are changed, so that the heights are computed for points midway between each pair of points computed at the previous level. This process is repeated at each level, so when the *For* loop is finally complete, heights have been computed for all of the members of the height array. Each height is related to the point heights around it, and includes a random roughness factor. The main program then calls *SaveHeightBuffer*, which saves the height array to a disk file database called *Mountain.Dat*. Figures 15-1, 15-2, and 15-3 are of typical mountains produced by databases created with this program using the z-buffering rendering technique.

```
{
  

Fractal Landscape Generator


}
```

Uses

Dos, Crt;

{\$I Math.Inc}

{\$I Graph.Inc}

FRACTAL PROGRAMS TO GENERATE DATABASES

```
{ $I Render.Inc }
```

```
Const
```

```
    NumberLevels = 7;
```

```
    MaximumLevel = 7;           { maximum number of levels }
```

```
    Roughness    = 100;        { landscape Roughness (feet) }
```

```
Var
```

```
    Points, Space : Integer;
```

```
Procedure Initialization;
```

```
    Var
```

```
        I : Integer;
```

```
    Begin
```

```
        Randomize;
```

```
        Points := 1;
```

```
        For I := 1 To NumberLevels Do  
            Points := Points * 2;
```

```
        Space := 1;
```

```
        For I := 1 To (MaximumLevel - NumberLevels) Do  
            Space := Space * 2;
```

```
        MaxHeight := Roughness;
```

```
        HeightBufferScalingFactor
```

```
    End;
```

```
Function Hgt(I, J : Integer) : Integer;
```

```
    Begin
```

```
        Hgt := Height[I * Space, J * Space]
```

```
    End;
```

```
Procedure CalculateFractalHeights;
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
Var
    I, J, K      : Integer;
    Deviation     : Integer;
    HalfDeviation : Integer;
    Skip, Offset  : Integer;
    X,Y,L,N      : Integer;

Procedure InterpolateHeight(I, J, I0, J0, I1, J1 : Integer);

Begin
    Height[I * Space, J * Space] := ( Hgt(I0, J0) + Hgt(I1,
        J1) ) Div 2 + Round(Random * Deviation - HalfDeviation)
End;

Begin
    WriteLn;
    HalfDeviation := Roughness;
    Offset := Points;
    For N := 1 To NumberLevels Do
        Begin
            Deviation := HalfDeviation; { Roughness / 2^n }
            HalfDeviation := Deviation Div 2; { Roughness / 2^n
                                                / 2 }

            Skip := Offset; { Points / 2^n }
            Offset := Skip Div 2; { Points / 2^n / 2 }
            Write('Pass Level ',N,' -> ');
            J := 0;
            Repeat
                I := Offset;
                Repeat
                    InterpolateHeight(I , J,
                        I - Offset, J,
                        I + Offset, J);
                    X := Points - I;
                    Y := Points - J;
                    InterpolateHeight(X , Y,
                        X + Offset, Y,
                        X - Offset, Y);
                    I := I + Skip
                Until (I > Points);
            Until (J > Points);
        End
    End
```


FRACTAL PROGRAMS TO GENERATE DATABASES

```
J := J + Skip
Until (J > Points);
Write(' X');
I := Points;
Repeat
  J := Offset;
  Repeat
    InterpolateHeight(I, J,
      I, J + Offset,
      I, J - Offset);
    X := Points - I;
    Y := Points - J;
    InterpolateHeight(X, Y,
      X, Y - Offset,
      X, Y + Offset);
    J := J + Skip
  Until (J > I);
  I := I - Skip
Until Not(I > 0);
Write(' Y');
I := 0;
Repeat
  J := Offset;
  K := Points - I;
  Repeat
    L := I + J;
    InterpolateHeight(L, J,
      L - Offset, J - Offset,
      L + Offset, J + Offset);
    X := Points - L;
    Y := Points - J;
    InterpolateHeight(X, Y,
      X + Offset, Y + Offset,
      X - Offset, Y - Offset);
    J := J + Skip
  Until (J > K);
  I := I + Skip
Until Not(I < Points);
WriteLn(' Diag')
End;
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
    WriteLn
  End;

Procedure ZeroHeightBufferBelowWaterLevel;

  Var
    I, J : Integer;

  Begin
    For I := 0 To Res Do
      For J := 0 To Res Do
        If Height[I,J] < 0 Then
          Height[I,J] := 0
      End;
    End;

  {


Main Program


  }
```

```
Begin
  Title;
  WriteLn('Mountain Scene Generator');
  WriteLn;
  WriteLn('By Chris Watkins');
  WriteLn;

  ClearHeightBuffer;
  Initialization;

  Res := Points;          { [ 2^level , 2^(level-1) ] }

  ObjectFile := 'Mountain';

  CalculateFractalHeights;
```


FRACTAL PROGRAMS TO GENERATE DATABASES

```
ZeroHeightBufferBelowWaterLevel;  
  
SaveHeightBuffer(ObjectFile);  
  
ExitGraphics  
End.
```

Figure 15-1. Listing of Program to Generate Mountain Databases

Because of the use of random numbers in this program, and because the function *Randomize* was called during initialization assuring that different new and random numbers are used at each pass through the program, a unique mountain scene will be produced from each run of the program. However, each time you run the program, the new database file will be written out as *Mountain.Dat*, overwriting the previous data in the file. Therefore, if you have a mountain scene that you want to keep, rename the *Mountain.Dat* file to something else before you run the *Mountain.Pas* file again.

Three-Dimensional Mandelbrot Sets

The program *CPM.Pas* creates the database for a three-dimensional Mandelbrot set. The program is listed in Figure 15-5 and begins with two sets of constants for creating two different Mandelbrot set databases. One is presently commented out. If you want to run this one, remove the curly brackets from around the second set of constants and put them around the first set of constants. The main program clears the screen, sets up some initial parameters, and calls *InitGraphics* to enter the graphics mode. The program then calls *ClearHeightBuffer* to zero out the height array. The program enters two nested *For* loops to iterate for every point on the two-dimensional display. For each iteration of the inner loop, a *While* loop iterates for the iterated Mandelbrot equation as many times as specified by *Iter* or until the square of the *xy* vector becomes greater than 20000. The equations in this loop are the same for the Mandelbrot set in Chapter 3. However, at the end of the *While* loop, instead of directly plotting a color as a function of the number of iterations, the program uses the exiting values of x^2 and y^2 to compute the value of *MSetPot*. This value is the continuous potential of the point with respect to the set, in the same sense that the potential of an electric field is calculated. It can also be scaled and used as a height

for a three-dimensional display. The program computes this height for the appropriate member of the height array. It also plots a point to the screen in the appropriate color for a two-dimensional display for reference as to how the plot is proceeding.

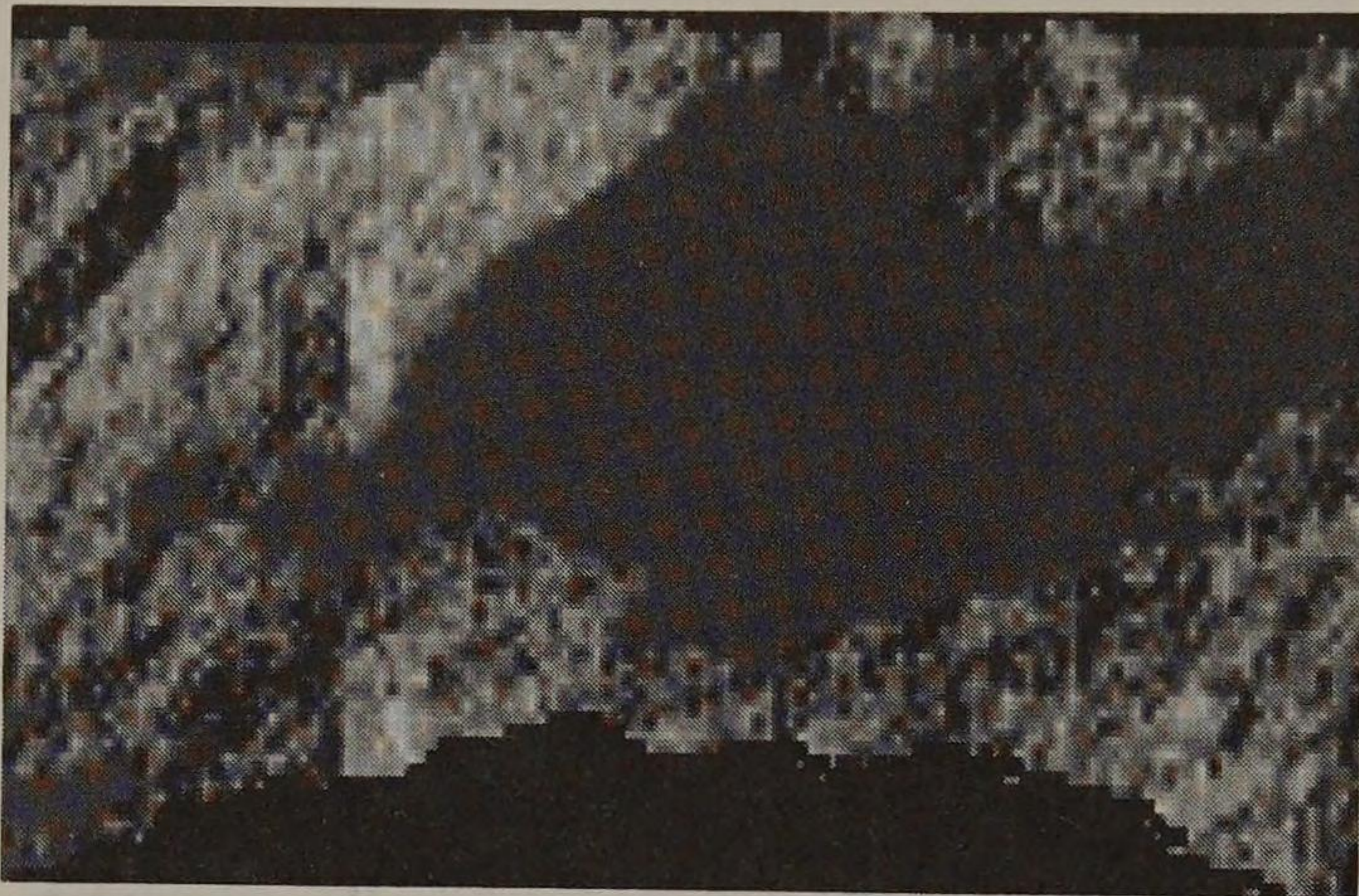


Figure 15-2. Fractal Mountain Created by Z-Buffer Rendering of *Mountan.Dat*

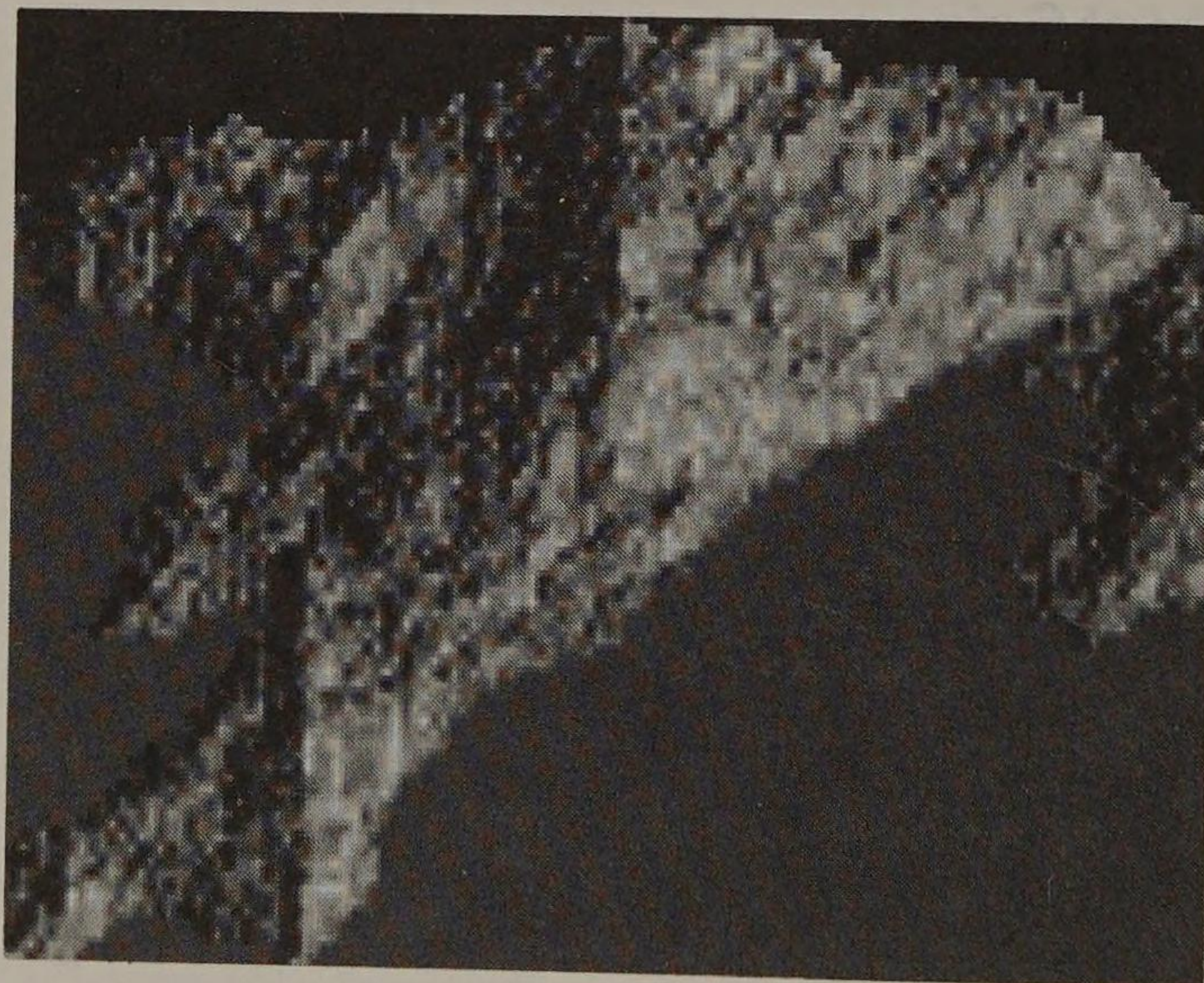


Figure 15-3. Fractal Mountain Created by Z-Buffer Rendering of *Mountan1.Dat*

The continuous method yields a potential that is smallest at the boundary of the Mandelbrot set and gets larger and larger as the distance of the point from the set boundary increases. This gives a three-dimensional display in which the set itself is down in a hole, with sides that slope sharply down to it. The display is more

interesting if the set is the plateau of a mountain, with the sides sloping upward. To achieve this effect, the program calls *InverseHeightBuffer* after the nested loops are complete. This procedure first determines the maximum height stored in the buffer, and then subtracts every member of the height array to create a new value for the member. As a result, the Mandelbrot set itself is at the maximum height and the points that previously were highest are now at zero. The program then calls *SaveHeightBuffer*, which stores the height array in a database file.

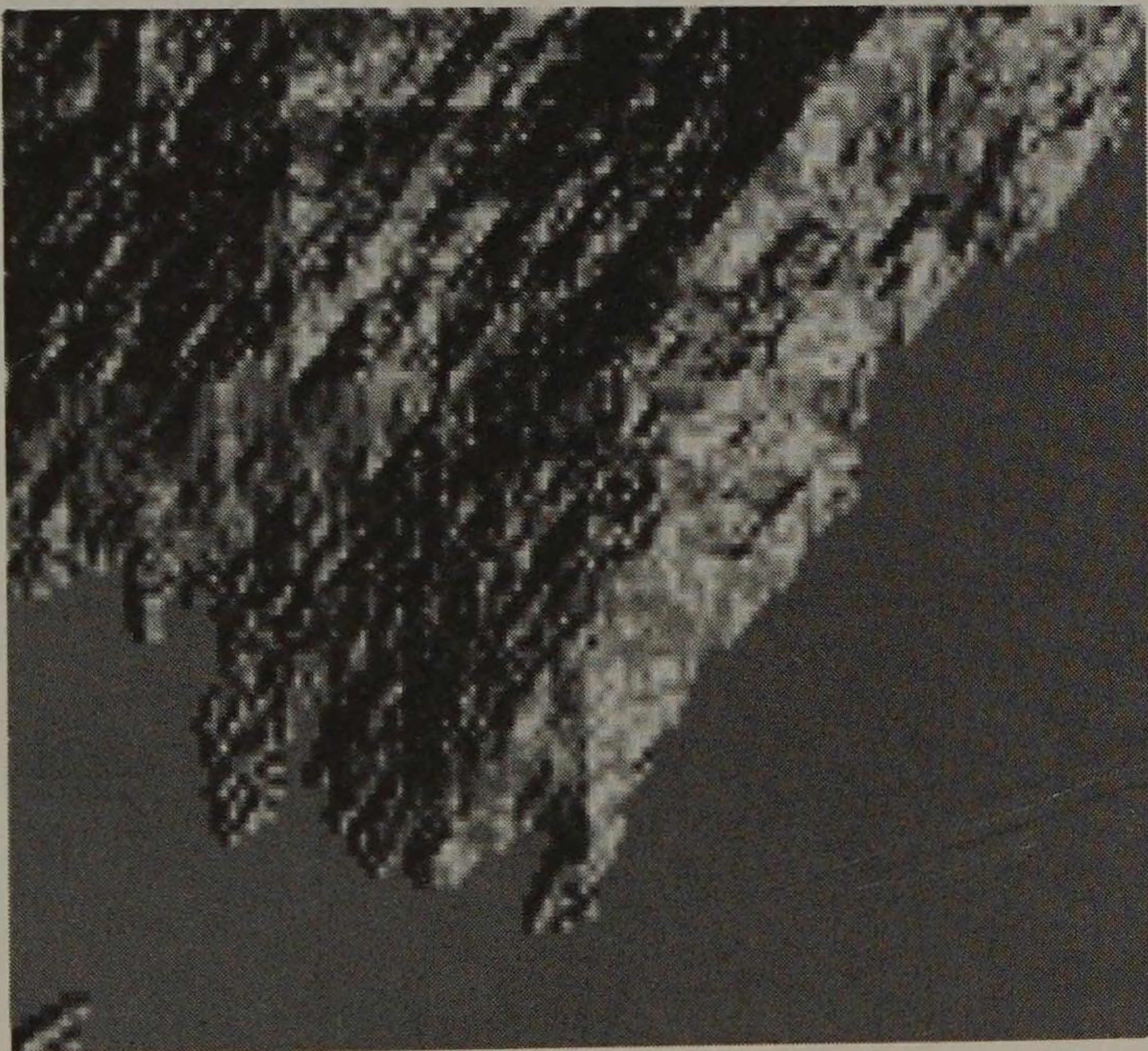


Figure 15-4. Fractal Mountain Created by Z-Buffer Rendering of *Mountain2.Dat*

Three-Dimensional Julia Sets

The program *CPMJ.Pas* creates the database for a three-dimensional Julia set. The program is listed in Figure 15-6 and begins with two sets of constants for creating two different Julia set databases. One is presently commented out. If you want to run this one, remove the curly brackets from around the second set of constants and put them around the first set. The program is similar to the Mandelbrot set, except that the variables in the iterated equation are different. The main program clears the screen, sets up initial parameters, and calls *InitGraphics* to enter the graphics mode.

The program then calls *ClearHeightBuffer* to zero out the height array. Next the program enters two nested *For* loops to iterate for every point on the two-dimensional display. For each iteration of the inner loop, a *While* loop is run which iterates for the iterated Julia set equation as many times as specified by *Iter* or until the square of the *xy* vector becomes greater than 20000. The equations in this loop are the same as for the Mandelbrot set in Chapter 3. However, at the end of the *While* loop, instead of directly plotting a color as a function of the number of iterations, the program uses the exiting values of x^2 and y^2 to compute the value of *MSetPot*. This value is the continuous potential of the point with respect to the set, in the same sense that the potential of an electric field is calculated. It can also be scaled and used as a height for a three-dimensional display. The program computes this height for the appropriate member of the height array. It also plots a point to the screen in the appropriate color for a two-dimensional display for reference.

```
{
  {
    The Continuous Potential Method for the
      Mandelbrot Set
  }
}
```

Uses

Dos, Crt;

{ \$I Math.Inc }

{ \$I Graph.Inc }

{ \$I Render.Inc }

{ 125 Iterations Maximum }

FRACTAL PROGRAMS TO GENERATE DATABASES

Const

```
ObjF = 'CPM1';  
XMin = -2.50;  
XMax = 1.35;  
YMin = -1.80;  
YMax = 1.80;  
Iter = 32;  
Scal = 32767;
```

{

Const

```
ObjF = 'CPM2';  
XMin = -0.19;  
XMax = -0.13;  
YMin = 1.01;  
YMax = 1.06;  
Iter = 125;  
Scal = 3276700;
```

}

Var

```
ix, iy : Integer;  
Iters : Integer;  
cx, cy : Real;  
dx, dy : Real;  
x, y : Real;  
x2, y2 : Real;  
temp : Real;  
powerof2 : Real;  
MSetPot: Real;
```

Procedure InverseHeightBuffer;

Var

```
i, j : Integer;  
Max : Integer;
```

Begin

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
Max := 0;
For i := 0 To MaxRes Do
  For j := 0 To MaxRes Do
    If (Height[i,j] > Max) Then Max := Height[i,j];
  For i := 0 To MaxRes Do
    For j := 0 To MaxRes Do
      Height[i,j] := Max - Height[i,j]
End;

{
  Main Program
}
```

```
Begin
  ClrScr;

  Scaling := 32767;

  Res := MaxRes;

  ObjectFile := ObjF;

  InitGraphics;

  ClearHeightBuffer;

  dx := (XMax - XMin) / (Res - 1);
  dy := (YMax - YMin) / (Res - 1);
  For iy := 0 To Res-1 Do
    Begin
      cy := YMin + iy * dy;
      For ix := 0 To Res-1 Do
        Begin
          cx := XMin + ix * dx;
          x := cx;
          x2 := Sqr(x);
          y := cy;
```


FRACTAL PROGRAMS TO GENERATE DATABASES

```

y2 := Sqr(y);
Iters := 0;
While (Iters < Iter) And (x2+y2 < 20000) Do
Begin
    temp := cx + x2 - y2;
    y := cy + 2 * x * y;
    y2 := Sqr(y);
    x := temp;
    x2 := Sqr(x);
    Iters := Succ(Iters)
End;
If (Iters < Iter) Then
    MSetPot := 0.5 * Log(x2+y2) / Power(2, Iters)
Else
    MSetPot := 0.0;
Height[ix,iy] := Round(MSetPot * Scal);
PutPixel(ix, iy, Iters Div MaxCol, Iters Mod
                                         MaxInten);
End
End;

InverseHeightBuffer;

SaveHeightBuffer(ObjectFile);

ExitGraphics
End.

```

Figure 15-5. Listing of Program to Generate Three-Dimensional Mandelbrot Sets

The continuous method yields a potential that is smallest at the boundary of the Julia set and gets larger as the distance between the point and the set boundary increases. This gives a three-dimensional display in which the set itself is down in a hole, with sides that slope sharply down to it. The display is more interesting if the set is a plateau on a mountain, with the sides sloping upward. To achieve this effect, the program calls the procedure *InverseHeightBuffer* after the nested loops are complete. This procedure duplicates the procedure described above for the Mandelbrot set. The program then calls *SaveHeightBuffer*, which stores the height array in a database file.

Fractals Using Quaternions

A quaternion is similar to a three-dimensional vector, except that it has a strange characteristic: Multiplication of quaternions does not follow the commutative law. For most types of numbers, the following expression is true:

$$a \times b = b \times a \quad (\text{Equation 15-1})$$

In other words, the order in which the numbers are multiplied doesn't matter; you'll get the same result no matter what order is used. This is known as the commutative law. For quaternions, this is not true. In fact, if a and b are quaternions, then:

$$ab = - (ba) \quad (\text{Equation 15-2})$$

With this, let's look at just what quaternions are. You may define a quaternion as consisting of a scalar q_0 , and a three-dimensional vector having components q_1 , q_2 , and q_3 , directed along the mutually orthogonal axes identified by the unit vectors i , j , and k . These vectors follow the rules for vector cross-products in a right-hand coordinate system, namely:

$$\begin{array}{ll} ij = k & ji = -k \\ jk = i & kj = -i \\ ki = j & ik = -j \end{array} \quad (\text{Equation 15-3})$$

and

$$ijk = i^2 = j^2 = k^2 = -1 \quad (\text{Equation 15-4})$$

You can write the quaternion as:

$$Q = q_0 + iq_1 + jq_2 + kq_3 \quad (\text{Equation 15-5})$$

or alternately as:

FRACTAL PROGRAMS TO GENERATE DATABASES

$$Q = q_0 + \underline{q} \quad (\text{Equation 15-6})$$

where the underlining means that the variable is a three-dimensional vector. In other words:

$$\underline{q} = iq_1 + jq_2 + kq_3 \quad (\text{Equation 15-7})$$

Quaternions have interesting uses in transforming three-dimensional systems. For instance, transforming from the coordinate system of a space shuttle to earth coordinate system without encountering singularities. They can also be used in iterative equations to create strange, four-dimensional fractals. Of course, you'll really be in deep trouble if you try to display a four-dimensional figure on a two-dimensional screen. What is usually done is to compute the data for a three-dimensional slice of the figure and then use the previously described methods for projecting this onto the two-dimensional screen.

{

The Continuous Potential Method for the
Julia Set

$c = p + qi$ ACP = attractive cycle period

p	q	
-----	-----	
0.31	0.04	-
-0.11	0.6557	- JSet ACP=3 before decay into Cantor set
-0.194	0.6557	- after decay into Cantor set
-0.12	0.74	-
0.0	1.0	- lightning dendrite
-0.74543	0.11301	- seahorse valley c-value
-1.25	0.0	- parabolic case $c > 1.25$ ACP=2, $c < 1.25$ ACP=4
-0.481762	-0.531657	-
-0.39054	-0.58679	-

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
-0.15652 -1.03225 - dendrite with beads (secondary
MSet)
0.11031 -0.67037 - Fatou dust
0.27334 0.00742 - parabolic case small c ACP=20
}
```

Uses

Dos, Crt;

{ \$I Math.Inc }

{ \$I Graph.Inc }

{ \$I Render.Inc }

{ 125 Iterations Maximum }

Const

ObjF = 'CPMJ1';

p = -0.12;

q = 0.74;

XMin = -1.4;

XMax = 1.4;

YMin = -1.4;

YMax = 1.4;

Iter = 32;

Scal = 32767;

{

ObjF = 'CPMJ2';

p = 0.31;

q = 0.04;

XMin = -1.4;

XMax = 1.4;

FRACTAL PROGRAMS TO GENERATE DATABASES

```
YMin = -1.4;  
YMax = 1.4;  
Iter = 32;  
Scal = 32767;
```

```
}
```

```
Var
```

```
  x, y : Real;  
  x2, y2 : Real;  
  temp : Real;  
  powerof2 : Real;  
  ix, iy : Integer;  
  Iters : Integer;  
  cx, cy : Real;  
  dx, dy : Real;  
  Z : Integer;  
  JSetPot: Real;
```

```
Procedure InverseHeightBuffer;
```

```
Var
```

```
  i, j : Integer;  
  Max : Integer;
```

```
Begin
```

```
  Max := 0;  
  For i := 0 To MaxRes Do  
    For j := 0 To MaxRes Do  
      If (Height[i,j] > Max) Then Max := Height[i,j];  
  For i := 0 To MaxRes Do  
    For j := 0 To MaxRes Do  
      Height[i,j] := Max - Height[i,j]
```

```
End;
```

```
{
```

Main Program

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
}  
  
Begin  
  ClrScr;  
  
  Scaling := 32767;  
  
  Res := MaxRes;  
  
  ObjectFile := ObjF;  
  
  InitGraphics;  
  
  ClearHeightBuffer;  
  
  dx := (XMax - XMin) / (Res - 1);  
  dy := (YMax - YMin) / (Res - 1);  
  For ix := 0 To (Res-1) Div 2 Do { symmetry about the origin }  
    Begin  
      cx := XMin + ix * dx;  
      For iy := 0 To Res-1 Do  
        Begin  
          cy := YMin + iy * dy;  
          x := cx;  
          x2 := Sqr(x);  
          y := cy;  
          y2 := Sqr(y);  
          Iters := 0;  
          While (Iters < Iter) And (x2+y2 < 20000) Do  
            Begin  
              temp := p + x2 - y2;  
              y := q + 2 * x * y;  
              y2 := Sqr(y);  
              x := temp;  
              x2 := Sqr(x);  
              Iters := Succ(Iters)  
            End;  
          If (Iters < Iter) Then  
            JSetPot := 0.5 * Log(x2+y2) / Power(2, Iters)  
          Else  
            JSetPot := 0.0;
```


FRACTAL PROGRAMS TO GENERATE DATABASES

```

        Z := Round(JSetPot * Scal);
        Height[ix,iy] := Z;
        Height[Res-1-ix,Res-1-iy] := Z;
        PutPixel(ix      , 199-iy      ,
                ITERS Div MaxCol, ITERS Mod MaxInten);
        PutPixel((Res-1-ix), 199-(Res-1-iy),
                ITERS Div MaxCol, ITERS Mod MaxInten)
    End
End;

InverseHeightBuffer;

SaveHeightBuffer(ObjectFile);

ExitGraphics
End.

```

Figure 15-6. Listing of program to generate three-dimensional Julia Sets.

Quaternion Mathematics

Quaternion addition and subtraction can be done on a component-by-component basis, just like vectors are added and subtracted, although quaternion multiplication gets a little more complicated. Suppose you have two quaternions, P and Q . The product is:

$$\begin{aligned}
 PQ &= (p_0q_0 + p_0\mathbf{q} + q_0\mathbf{p} + i^2p_1q_1 + j^2p_2q_2 + k^2p_3q_3 + \\
 &\quad i p_1(jq_2 + kq_3) + j p_2(iq_1 + kq_3) + k p_3(iq_1 + jq_2)) \\
 &= p_0q_0 + p_0\mathbf{q} + q_0\mathbf{p} - (\mathbf{p}\cdot\mathbf{q}) + \mathbf{p} \times \mathbf{q} \quad (\text{Equation 15-8})
 \end{aligned}$$

where $\mathbf{p}\cdot\mathbf{q}$ and $\mathbf{p} \times \mathbf{q}$ are the vector dot product and vector cross product, respectively as defined in any text on vector analysis. You will see that this expression has two scalar terms (which may be summed to give one scalar) and three, three-dimensional vector terms (which may be summed to give one three-dimensional vector). The result consists of a scalar and three-dimensional vector (which is a quaternion), so that the product of two quaternions is a quaternion. Now, since one of these terms is

not commutative ($p \times q = -q \times p$), the quaternion multiplication is also not commutative.

There is no need to go into any further detail on quaternion mathematics, except to say that you need a complete set of functions to perform these operations before beginning any serious programming with quaternions. Such a set of functions is provided in the file *Quater.Inc*, listed in Figure 15-7. The functions are fairly self-explanatory; by examining them in detail you can get a good idea of what is involved in each operation.

Generating Quaternion Fractal Databases

The file *Quater.Pas*, listed in Figure 15-8, includes the main program and procedures for generating databases for quaternion fractals. The main program is relatively simple. It sets the maximum height and then calls *HeightScalingFactor* to determine the scaling factor for the height values, and then calls the procedure *Parameters* to establish some initial parameters. Next, it calls *ScanSpace* which does the work necessary to fill the height array with height values. The program finally calls *SaveHeightBuffer* to save the height data to a disk database file.

The procedure *Parameters* first sets the text color to yellow and the background to blue, then clears the display screen and displays ten lines of data about the quaternion generator. Next, it calls *ClearHeightBuffer* to zero the height array, then calls *QSetCons*. This procedure sets up a number of constants needed by the program. There are two versions of this procedure, one to generate a shape similar to the two-dimensional dragon curves, with data stored in the file *Dragon.Dat*, and the other to generate a shape similar to two-dimensional Julia sets. This version creates the data file *RevSolid.Dat*. As shown, the version for the dragon is commented out. If you want to create this database, remove the comment symbols ((* and *)) and place them around the alternate version of the procedure. The *Parameters* procedure then sets up scale factors in the x , y , and z directions and offsets in the x and y directions. The procedure *ScanSpace* begins by calling *MaxDepthOfRecursion*. Use a recursive technique to determine height values, which will determine the number of recursions needed to get from the stated display size down to pixel levels of resolution. The resulting number is displayed on the screen, and stored in the parameter *MaxDepth*.

ScanSpace then computes the change on x and y required for each increment in the height array, and the change in z for each increment of allowed height. The procedure enters a pair of nested *Repeat* loops which iterate once for every member of the height array. Within these loops, the procedure sets up the initial value for the quaternion P (at the farthest z) and then calls *Iterate*. This procedure repeatedly calls *Func* (which performs one iteration of the quaternion equation) until either the maximum number of iterations is achieved, or the magnitude of the quaternion exceeds a predetermined level. When *Iterate* returns to *ScanSpace*, if the maximum number of iterations are not reached, you are done and the proper height value is determined. In this case, you call *UpdateHeight* which places the height data in the proper member of the height array. If the maximum number of iterations was reached, the procedure runs again for the nearest z . If the maximum number of iterations is reached you are done and the proper height value is determined, so you can again call *UpdateHeight* to place the height data in the proper member of the height array. If neither of the above conditions occurred, call *RecursiveCalc*. This procedure sets up new limits for the beginning and ending z values and runs *Iterate* to perform the tests all over again, either terminating when the correct height value is found or running *RecursiveCalc* again, until finally the procedure zeroes in on the correct height value. When all of this has been done for one slice of data (a column), display "Slice # nn ," where nn is the column number, and then increment *Slice*, set X to a new value, and proceed through the loop again, until the entire height array has been filled and the procedure returns to the main program.

{

Quaternion Mathematics Routines

QAdd - quaternion addition
 QSubtract - quaternion subtraction
 QMultiply - quaternion multiplication
 QDivide - quaternion division
 QSquareRoot - quaternion square root
 QSquare - quaternion square

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

$$Q = R + Ai + Bj + Ck = q[0] + q[1]i + q[2]j + q[3]k$$

}

{ r = p + q }

Procedure QAdd(p, q : FDA; Var r : FDA);

Begin

r[0] := p[0] + q[0];

r[1] := p[1] + q[1];

r[2] := p[2] + q[2];

r[3] := p[3] + q[3]

End;

{ r = p - q }

Procedure QSubtract(p, q : FDA; Var r : FDA);

Begin

r[0] := p[0] - q[0];

r[1] := p[1] - q[1];

r[2] := p[2] - q[2];

r[3] := p[3] - q[3]

End;

{ r = p x q }

Procedure QMultiply(p, q : FDA; Var r : FDA);

Begin

r[0] := p[0] * q[0] - p[1] * q[1] - p[2] * q[2] - p[3] *
q[3];

r[1] := p[0] * q[1] + p[1] * q[0] + p[2] * q[3] - p[3] *
q[2];

r[2] := p[0] * q[2] + p[2] * q[0] + p[3] * q[1] - p[1] *
q[3];

r[3] := p[0] * q[3] + p[3] * q[0] + p[1] * q[2] - p[2] *
q[1]

End;

FRACTAL PROGRAMS TO GENERATE DATABASES

{ $r = p / q$ }

Procedure QDivide(p, q : FDA; Var r : FDA);

Var

t, s : FDA;

a : Real;

Begin

q[1] := -q[1];

q[2] := -q[2];

q[3] := -q[3];

QMultiply(p, q, t);

QMultiply(q, q, s);

a := 1 / s[0];

r[0] := t[0] * a;

r[1] := t[1] * a;

r[2] := t[2] * a;

r[3] := t[3] * a

End;

{ $s = \sqrt{q}$ }

Procedure QSquareRoot3D(q : FDA; Var s : FDA);

Var

len, l, m : Real;

A, B : Real;

r : FDA;

Begin

len := Sqrt (Sqr(q[0]) + Sqr(q[1]) + Sqr(q[2]));

l := 1 / len;

r[0] := q[0] * l;

r[1] := q[1] * l;

r[2] := q[2] * l;

r[3] := 0;

m := 1 / Sqrt(Sqr(r[0]) + Sqr(r[1]));

A := Sqrt((1 + r[2]) / 2);

B := Sqrt((1 - r[2]) / 2);

s[0] := Sqrt(len) * B * r[0] * m;

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
s[1] := Sqrt(len) * B * r[1] * m;  
s[2] := Sqrt(len) * A;  
s[3] := q[3]  
End;  
  
{ r = q^2 }  
Procedure QSquare(q : FDA; Var r : FDA);  
  
  Var  
    a : Real;  
  
  Begin  
    a := 2 * q[0];  
    r[0] := Sqr(q[0]) - Sqr(q[1]) - Sqr(q[2]) - Sqr(q[3]);  
    r[1] := a * q[1];  
    r[2] := a * q[2];  
    r[3] := a * q[3]  
  End;
```

Figure 15-7. Listing of Quaternion Mathematics Routines

```
{  
  Quaternion Fractal Generator  
  
   $f(q) = q * q * \text{lambda}$   
  
  Iteration of the Forward Equation in  
  Quaternion Space  
  
  By Christopher D. Watkins  
}
```

```
Uses  
  Crt;
```


FRACTAL PROGRAMS TO GENERATE DATABASES

```
{ $I Math.Inc }
```

```
{ $I Render.Inc }
```

```
{ $I Quater.Inc }
```

```
Var
```

```

Q, L                : FDA;
DivergenceBoundary  : Integer;
MaximumIterations   : Integer;
XLeft , XRight      : Real;
YBottom, YTop       : Real;
ZBack , ZFront      : Real;

```

```
{
```

Constants for Quaternion Generator

```

+ QSetCons - setup constants for quaternion generator
}

```

```
(*
```

```
Const
```

```
ObjF = 'Dragon'; { DRAGON }
```

```
Procedure QSetCons;
```

```
Begin
```

```

Q[0] := 0.0; { quaternion variable Q = Q[0] + Q[1]i +
                                                    Q[2]j + Q[3]k }

```

```
Q[1] := 0.0;
```

```
Q[2] := 0.0;
```

```
Q[3] := 0.0;
```

```

L[0] := 0.2809; { and constant Lambda = L[0] + L[1]i +
                                                    L[2]j + L[3]k }

```

```
L[1] := 0.53;
```

```
L[2] := 0.0;
```

```
L[3] := 0.0;
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
DivergenceBoundary := 4; { This is the Square of
                               Magnitude of Zc }
MaximumIterations := 100; { Number of Iterations per
                               Point }

XLeft := -1.3;      { Define Region for Examination }
XRight := 1.3;
YBottom := -1.3;
YTop := 1.3;
ZBack := 0.0;
ZFront := 2.0
End;
*)

Const
  ObjF = 'RevSolid'; { SOLID OF REVOLUTION }

Procedure QSetCons;

Begin
  Q[0] := 0.0; { quaternion variable Q = Q[0] + Q[1]i +
                               Q[2]j + Q[3]k }
  Q[1] := 0.0;
  Q[2] := 0.0;
  Q[3] := 0.0;
  L[0] := -1.0; { and constant Lambda = L[0] + L[1]i + L[2]j
                               + L[3]k }
  L[1] := 0.0;
  L[2] := 0.0;
  L[3] := 0.0;
  DivergenceBoundary := 4; { This is the Square of
                               Magnitude of Zc }
  MaximumIterations := 100; { Number of Iterations
                               per Point }
  XLeft := -1.8;      { Define Region for Examination }
  XRight := 1.8;
  YBottom := -1.8;
  YTop := 1.8;
  ZBack := 0.0;
  ZFront := 1.8
```


FRACTAL PROGRAMS TO GENERATE DATABASES

End;

Var

XScale, YScale, ZScale : Real;
XOffset, YOffset : Real;

Procedure Parameters;

Begin

TextColor(Yellow);
TextBackground(Blue);
ClrScr;
WriteLn('Quaternion Generator');
WriteLn;
WriteLn('Equation : $f(q) = \lambda + q * q$ ');
WriteLn;
WriteLn('lambda : quaternion constant for a particular
fractal curve');
WriteLn('q : quaternion variable seeded as 0+0i+0j+0k');
WriteLn;
WriteLn('The fractal surface is realized by examination of
the');
WriteLn(' divergence properties of the equation in
quaternion space');
WriteLn;
ClearHeightBuffer; { clear the object space }
QSetCons;
XScale := Res / (XRight - XLeft);
XOffset := -XScale * XLeft;
YScale := Res / (YBottom - YTop);
YOffset := -YScale * YTop;
ZScale := Res * Scaling / (ZFront - ZBack)

End;

Procedure Func(Q : FDA; Var NQ : FDA);

Begin

QSquare(Q, Q); { $q \leq q^2 + \lambda$ }

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
    QAdd(Q, L, NQ)
End;
```

```
Procedure Iterate(P : FDA; Var NumberOfIterations : Integer);
```

```
    Var
        NQ : FDA;
```

```
Begin
```

```
    Func(P, Q);
```

```
    NumberOfIterations := 0;
```

```
    Repeat
```

```
        Func(Q, NQ);
```

```
        NumberOfIterations := Succ(NumberOfIterations);
```

```
        Q[0] := NQ[0];
```

```
        Q[1] := NQ[1];
```

```
        Q[2] := NQ[2];
```

```
        Q[3] := NQ[3];
```

```
    Until Not(Sqr(NQ[0]) + Sqr(NQ[1]) < DivergenceBoundary) Or
           (NumberOfIterations = MaximumIterations)
```

```
End;
```

```
Procedure ScanSpace;
```

```
    Procedure UpdateHeight(X, Y, Z : Real);
```

```
        Var
```

```
            PlotX, PlotY, PlotZ : Integer;
```

```
        Begin
```

```
            PlotX := Round( XScale * X + XOffset );
```

```
            PlotY := Round( YScale * Y + YOffset );
```

```
            PlotZ := Round( ZScale * Z );
```

```
            Height[PlotX, PlotY] := PlotZ
```

```
        End;
```

```
Var
```

```
    MaxDepth : Byte;
```


FRACTAL PROGRAMS TO GENERATE DATABASES

Procedure CalculateMaxDepthOfRecursion;

Var

k : Real;

Begin

k := Res * 0.5; { recursion is seeded with res / 2 }

MaxDepth := 0;

Repeat { MaxDepth will indicate how many recursions
are }

k := k * 0.5; { necessary to obtain the proper pixel
resolution }

MaxDepth := MaxDepth + 1

Until (k < 1); { 1 is the size of a pixel }

WriteLn('Recursion Depth based on Resolution is
' ,MaxDepth)

End;

Var

P : FDA;

X, Y, Z : Real;

NumberOfIterations : Integer;

Procedure RecursiveCalc(Zrec, Delta : Real; Depth : Integer);

Begin

Z := Zrec;

If Not(Depth = MaxDepth) Then

Begin

P[0] := X;

P[1] := Y;

P[2] := Z;

P[3] := 0;

Iterate(P, NumberOfIterations);

If (NumberOfIterations < MaximumIterations) Then

Z := Z - Delta

Else

Z := Z + Delta;

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
RecursiveCalc(Z, Delta * 0.5, Depth + 1) {  
    successive approximation }  
End  
End;  
  
Var  
    Dx, Dy, Dz      : Real;  
    Slice           : Integer;  
    Zseed, DZseed   : Real;  
  
Label  
    Done;  
  
Begin  
    CalculateMaxDepthOfRecursion;  
  
    Dx := ( XRight - XLeft ) / Res;  
    Dy := ( YBottom - YTop ) / Res;  
    Dz := ( ZFront - ZBack );  
    Zseed := Dz * 0.5;  
    DZseed := Zseed * 0.5;  
  
    Slice := 0;  
  
    X := XLeft;  
  
    Repeat  
        Y := YTop;  
  
        Repeat  
            Z := ZBack; { check back - if Iters < MaxIters then  
                           next location }  
            P[0] := X; { because there will be nothing above  
                           ZBack }  
            P[1] := Y; { (for our purposes) }  
            P[2] := Z;  
            P[3] := 0;
```


FRACTAL PROGRAMS TO GENERATE DATABASES

```
Iterate(P, NumberOfIterations);
If (NumberOfIterations < MaximumIterations) Then
    GOTO Done;

Z := ZFront;    { check front - if Iters >= MaxIters
                  then next location }

P[0] := X;    { because there is nothing in front of
               Zfront }

P[1] := Y;
P[2] := Z;
P[3] := 0;
Iterate(P, NumberOfIterations);
If Not(NumberOfIterations < MaximumIterations) Then
    GOTO Done;
RecursiveCalc( Zseed, DZseed, 0 );

{ use recursion to create a binary breakdown of
  iterations }
{ to succesively approximate z in order to increase
  speed }

Done: UpdateHeight(X, Y, Z);

Y := Y + Dy

Until ( Y < YBottom );

GotoXY(1,20);
WriteLn('Slice # ', Slice);
Slice := Slice + 1;

X := X + Dx

Until( X > XRight )
End;
```



```

{
  Main Program
}

Begin
  Res := MaxRes;

  ObjectFile := ObjF;

  MaxHeight := Res;
  HeightBufferScalingFactor;

  Parameters;

  ScanSpace;

  SaveHeightBuffer(ObjectFile)
End.

```

Figure 15-8. Listing of Program to Generate Quaternion Database

Mountains from Plasmas

The *Plasma.Pas* program listed in Figure 15-9 selects four randomly colored points at the corners of a display area. It then chooses a color for the midpoint between each pair of points, based on the average of the two endpoint colors plus a random factor. This sub-division process continues until every pixel within the display area has been colored. The resulting display is a plasma-like effect. The colors then cycle until a key is pressed. The original display is saved to provide a height buffer file for running of the z-buffering render program. The result of z-buffering is a mountain-like display.

FRACTAL PROGRAMS TO GENERATE DATABASES

```
{
  Plasma Generator by Christopher D. Watkins
}
```

Uses

Dos, Crt;

{\$I Math.Inc}

{\$I Graph.Inc}

{\$I Render.Inc}

Procedure InitPalette3(Var Color : PaletteRegister);

Const

mcol = 63;

Var

i : Byte;

Begin

Color[0].Red := 0;

Color[0].Grn := Round(mcol / 85);

Color[0].Blu := mcol;

For i := 1 To 85 Do

Begin

Color[i].Red := 0;

Color[i].Grn := Round((i * mcol) / 85);

Color[i].Blu := Round(((86-i) * mcol) / 85);

Color[i+85].Red := Round((i * mcol) / 85);

Color[i+85].Grn := Round(((86-i) * mcol) / 85);

Color[i+85].Blu := 0;

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
        Color[i+170].Red := Round( ( (86-i) * mcol) / 85);
        Color[i+170].Grn := 0;
        Color[i+170].Blu := Round( (i * mcol) / 85);
    End;
End;
```

```
Procedure UpdateHeight(x, y, Color : Integer);
```

```
    Begin
        Plot(x, y, Color And 255);
        Height[Res-X,Res-Y] := Color ShR 1
    End;
```

```
Procedure NewCol(xa, ya, x, y, xb, yb : Integer);
```

```
    Var
        Color : Integer;

    Begin
        Color := Abs(xa - xb) + Abs(ya - yb);
        Color := Random(Color ShL 1) - Color;
        Color := Color + (GetPixel(xa, ya) + GetPixel(xb, yb) + 1)
    ShR
        1;
        If (Color < 1) Then
            Color := 1
        Else
            If (Color > 255) Then
                Color := 255;
            If GetPixel(x,y) = 0 Then
                UpdateHeight(x, y, Color)
    End;
```

```
Procedure Subdivide(x1, y1, x2, y2 : Integer);
```

```
    Var
        x, y, Color : Integer;
```


FRACTAL PROGRAMS TO GENERATE DATABASES

```
Begin
  If Not ( (x2-x1 < 2) And (y2-y1 < 2) ) Then
    Begin
      x := (x1 + x2) ShR 1;
      y := (y1 + y2) ShR 1;
      NewCol(x1, y1, x, y1, x2, y1);
      NewCol(x2, y1, x2, y, x2, y2);
      NewCol(x1, y2, x, y2, x2, y2);
      NewCol(x1, y1, x1, y, x1, y2);

      Color := (GetPixel(x1, y1) + GetPixel(x2, y1) +
        GetPixel(x2, y2) + GetPixel(x1, y2) + 2) ShR 2;

      UpdateHeight(x, y, Color);

      Subdivide(x1, y1, x, y);
      Subdivide(x, y1, x2, y);
      Subdivide(x, y, x2, y2);
      Subdivide(x1, y, x, y2);
    End
  End;
End;
```

```
{
  

MAIN PROGRAM


}
```

```
Var
  PalArray : PaletteRegister;
```

```
Begin
  ClearHeightBuffer;
```

```
  Res := MaxRes;
```

```
  ObjectFile := 'Plasma';
```

```
  MaxHeight := Res;
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
HeightBufferScalingFactor;  
  
InitGraphics;  
  
InitPalette3(PalArray);  
SetPalette(PalArray);  
  
Randomize;  
  
UpdateHeight( 0, 0, 255); { Make plasma low in front and high  
                           in back }  
UpdateHeight(Res, 0, 255); { so that mountain has this  
                           characteristic }  
UpdateHeight(Res, Res, 1);  
UpdateHeight( 0, Res, 1);  
  
Subdivide(0, 0, Res, Res);  
  
SaveHeightBuffer(ObjectFile);  
  
Repeat  
    CyclePalette(PalArray)  
Until KeyPressed;  
  
ExitGraphics  
End.
```

Figure 15-9. Listing of *Plasma.Pas* Program

Part IV

CHAPTER 16

Ray Tracing Fundamentals

RAY TRACING

Ray Tracing Fundamentals

Ray-tracing creates two-dimensional pictures of a three-dimensional world using a computer process that is the opposite of what a camera does in taking a picture. Using a camera, reflected light from every point in a scene projects through the camera lens to the film plane (the viewing screen) onto a permanent record (film). With ray-tracing, you postulate a ray from the viewer's eye that intersects the screen and continues in a straight line until it hits some object in the scene. If the object is reflective, continue the trace reflecting off the object at the proper angle until every source of reflected light has been accounted for that contributes to the object color. The resulting color transfers to the viewing screen at the point where the ray intersects it. When you do this for every point on the viewing screen, we have a realistic two-dimensional picture, complete with shadows and reflections. The limitations of this technique are that you must specify the scene in terms of primitive objects (spheres, parallelograms, triangles, fractals, etc.) and describe them with fairly simple equations. These, together with patterns projected on surfaces, light sources, and the position of the observer, provide the information needed to define a scene for the ray-tracer to trace every beam of light, creating the complex shadows and reflections that result in breathtaking realism.

To understand this, let's postulate a scene of various primitive objects and light sources. An observer, stationed at a specified point in the coordinate system is looking at this scene through the translucent screen of a color monitor. Each tiny point on the screen, corresponding to a pixel on this monitor, appears to have some color associated with it. The color actually is the color of the ray of light that intersects the screen and projects onward to the observer's eye. This is the way a camera works. However, it is inefficient for the computer to construct light rays from objects because millions of rays from each object in the scene do not intersect the screen and continue to the viewer's eye. They are not really needed for picture

creation. Therefore, you trace the light ray from observer to screen in a straight line to the nearest object, considering only those rays essential for the picture. The color of the ray at the intersection with this object is the color that appears on the screen. You'll end up building a color data file assigning that pixel on the screen to the light ray color. The color of the ray at the object-intersection is not something that can be determined casually, since it often depends not only on the inherent color of the object, but also upon what happens to the light ray before it reaches the object. If the light is totally emitted by the object and a specific color, your problem is solved. This seldom is the case however, more often external light sources, and/or ambient light, illuminate the object. If the surface is transmitting a color from within a partially transparent object, reflecting a color from somewhere else, or changing a color originally projected by a light source, follow the path of the light-ray back toward the light sources and sum the various color contributions to determine the actual ray-color at the object intersection.

This is a very complex task, but not beyond the capabilities of your PC. Figure 16-1 shows the geometry of a simple, ray-traced scene. This is an example of how the process works. Now look at Figure 16-2, where a few, additional objects and light sources have been added with shadows and reflections. It's already complex, and when you consider that a realistic scene may contain thousands of objects, you begin to get an idea of the problem facing the computer. Fortunately, you only have to enumerate the primitive objects of which the scene is composed. All of the complicated interactions of light and shadow are performed by the computer program.

You can perform all ray-tracing tasks within the capacities of an IBM PC, or compatible, in a reasonable amount of time; so with the proper software, you will be in the picture-generating business. We are constrained, however, by the fact of defining a scene in terms of primitive, geometric objects. Each object has unique properties you must be aware of to perform accurate ray-tracing computations. For example, the normal for a sphere is different from the normal for a triangle. The point of intersection of the light ray and sphere requires a particular set of calculations. The calculations of the intersection of the light ray with a parallelogram are the same for all parallelograms, but different from those for a sphere. Note a couple of important

facts. First, the computer operations are computation-intensive. Second, the methodology is highly object-oriented. In other words, our orientation in creating the program is to specify things done to objects, and let the programming language address each type of object for each operation and behave appropriately. However, as the authors have tried to create code for all the most recent versions of Turbo Pascal, without too many modifications, they haven't used the latest object-oriented capabilities of Turbo Pascal 5.5 and 6.0. If you have one of these compilers, try modifying the ray-tracing program to work in an object-oriented mode. Initially, try running the program in its original state and become familiar with specifying scenes the ray-tracer translates into life-like pictures.

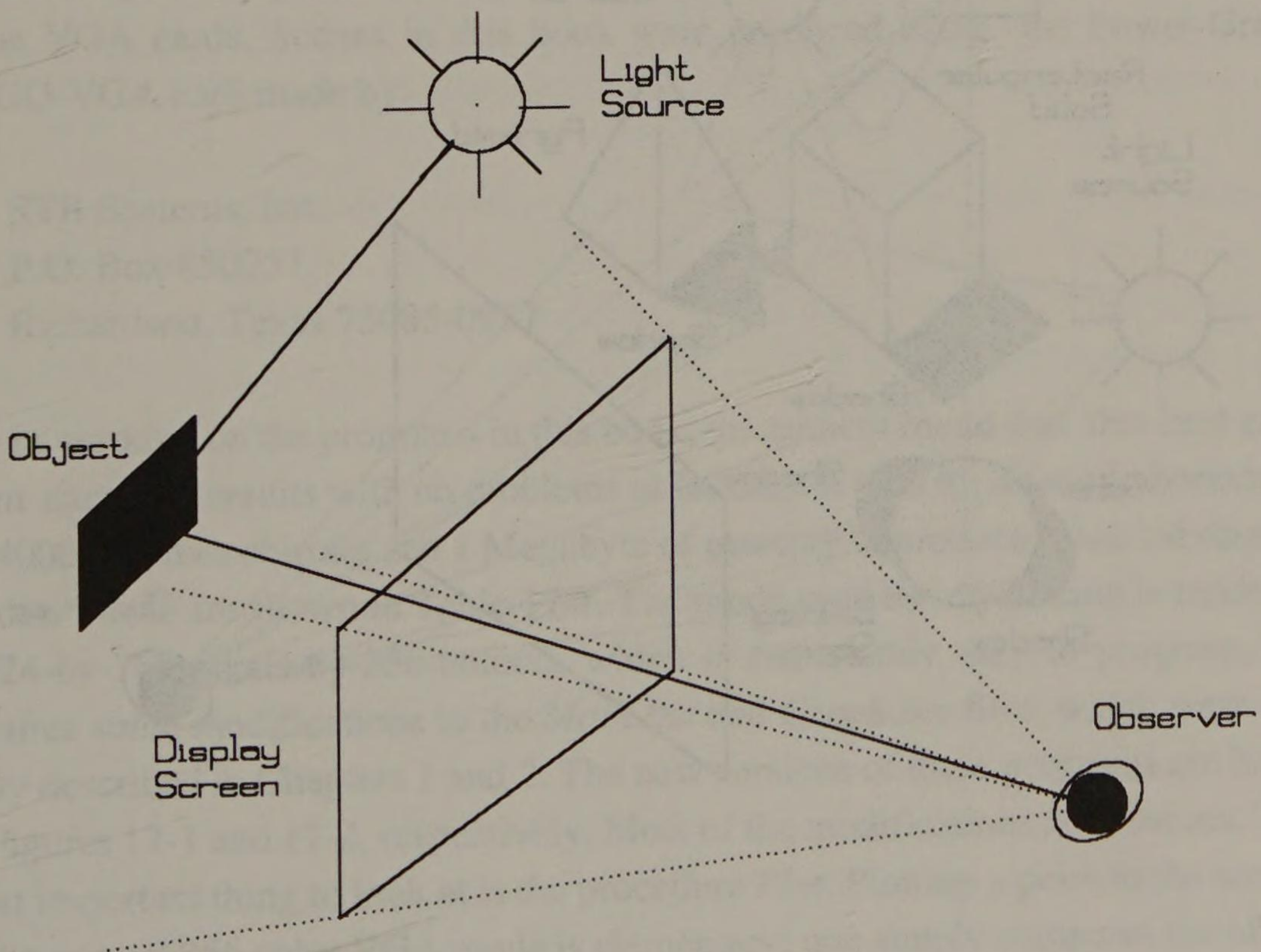


Figure 16-1. Ray Tracing Geometry

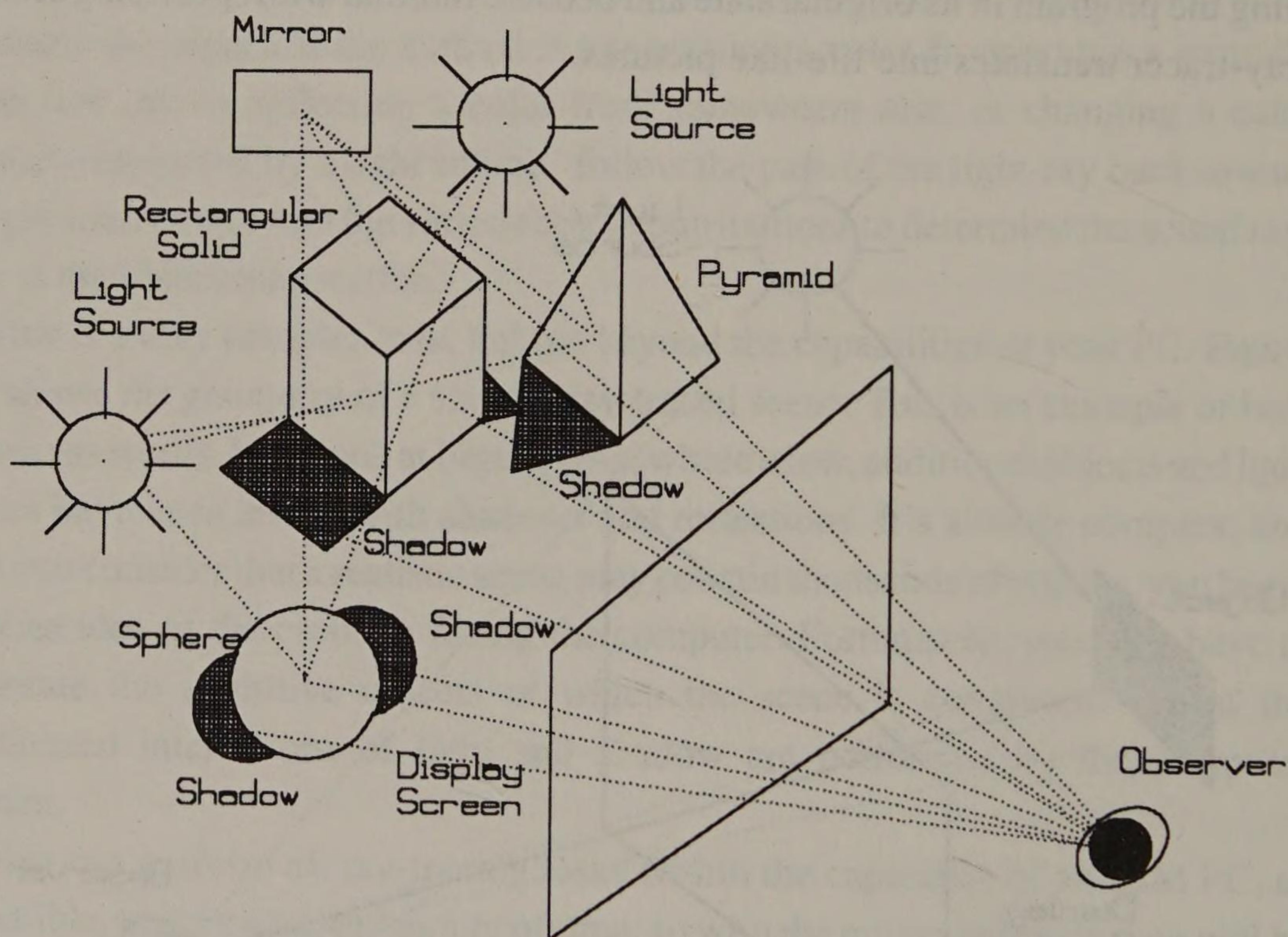


Figure 16-2. A More Complex Ray Tracing Diagram

High-resolution Graphics

With ray-tracing and a little ingenuity in displaying the picture, you can produce some fine-looking scenes using the 320-by-200-pixel-by-256-color mode of the VGA card. However, if you want to see some breathtaking scenes, you need to produce ray-traced pictures using one of the extended modes now available with some VGA cards. Scenes in this book were produced using the Power-Graph ERGO-VGA card made by:

STB Systems, Inc.

P.O. Box 850957

Richardson, Texas 75085-0957

In working on the programs in this book, the authors found that this card gave them excellent results with no problems or failures. It uses the Tseng Laboratories ET4000 graphics chip set and 1 Megabyte of memory to produce extended display modes. These are shown in Table 17-1. The mode used for ray-tracing is mode 56 (1024-by-768-pixels-by-256-colors), which is remarkably easy to program, but requires some modifications to the *Math.Inc* and *Graph.Inc* files, which were initially described in Chapters 1 and 2. The new versions of these programs are listed in Figures 17-1 and 17-2, respectively. Most of the modifications are obvious. The most important thing to look at is the procedure *Plot*. Plotting a point to the screen in the normal 256 color VGA mode is elementary; one simply computes the offset from the base-display memory address by counting the number of bytes required to get to the location of the desired pixel (at one byte per pixel) and writes out the color data to that address. The high-resolution mode is a little more complex, since all of the memory required to define a high-resolution display will not fit within the

memory address space allocated by the IBM PC memory scheme. Consequently, this memory is accessible through a 64K window, using a page register on the VGA board to track which part of the display memory it is. Thus the new Plot procedure computes the memory address offset like the old function, except that it is computed modulo 64K, to ensure fitting within the window. The procedure computes a page number, and sends it to the register in the VGA which stores it in order to select the proper memory segment. The color is written out to the memory location as before. As you can see, the penalty for using a high-resolution display is minimal. If you want to speed this procedure a bit, you can save the previous page number and not output to the page number register unless the new page number is different from the previous one.

Mode	Type	Resolution Characters	Resolution Pixels	Colors	Horizontal Frequency
0,1	Text	40 x 25	320 x 200	16	31.5 KHz
2,3	Text	80 x 25	640 x 200	16	31.5 KHz
4,5	Graphics	40 x 25	320 x 200	4	31.5 KHz
6	Graphics	80 x 25	640 x 200	2	31.5 KHz
7	Text	80 x 25	720 x 350	4	31.5 KHz
8	Text	132 x 25	1056 x 350	16	31.5 KHz
10	Text	132 x 44	1056 x 616	16	31.5 KHz
13	Graphics	40 x 25	320 x 200	16	31.5 KHz
14	Graphics	80 x 25	640 x 200	16	31.5 KHz
15	Graphics	80 x 25	640 x 350	2	31.5 KHz
16	Graphics	80 x 25	640 x 350	16	31.5 KHz
17	Graphics	80 x 30	640 x 480	2	31.5/38.0 KHz
18	Graphics	80 x 30	640 x 480	16	31.5/38.0 KHz
19	Graphics	40 x 25	320 x 200	256	31.5 KHz
34	Text	132 x 44	1056 x 616	16	31.5 KHz
35	Text	132 x 25	1056 x 350	16	31.5 KHz
36	Text	132 x 28	1056 x 400	16	31.5 KHz

Table 17-1. Display Modes for PowerGraph ERGO Extended VGA

Mode	Type	Resolution Characters	Resolution Pixels	Colors	Horizontal Frequency
36	Text	132 x 28	1056 x 400	16	31.5 KHz
41	Graphics	100 x 43	800 x 600	16	35.5/45.0 KHz
45	Graphics	80 x 25	640 x 350	256	31.5 KHz
46	Graphics	80 x 30	640 x 480	256	31.5/38.0 KHz
48	Graphics	100 x 43	800 x 600	256	35.5/45.0 KHz
55	Graphics	128 x 54	1024 x 768	16	35.5/48/57 KHz
56	Graphics	128 x 54	1024 x 768	256	35.5/48/57 KHz
106	Graphics	100 x 43	800 x 600	16	35.5/45.0 KHz
120	Graphics	80 x 25	640 x 400	256	31.5 KHz

Table 17-1. Display Modes for PowerGraph ERGO Extended VGA (continued)

One other thing to note is that the pixels in the 1024 x 768 mode are square, whereas in the 320 x 200 mode they are not. In other words, with a 3 x 4-screen aspect ratio, the horizontal distance per pixel with 1024 pixels in a row is the same as the vertical distance per pixel with 768 pixels in a column. This is not true of the 320 x 200 mode, where the distance of a horizontal pixel is greater than that of a vertical pixel. Thus in the high-resolution mode, when drawing circles, for example, no adjustment in the radius is necessary to obtain a true circle. In the low-resolution mode, the radius used in computing the horizontal direction must be different from that used in the vertical direction; otherwise you will get an oval instead of a circle. You'll see some corrections at various places in the graphics routines to take this problem into account.

Mathematical Functions

- Radians - converts degrees to radians
- Degrees - converts radians to degrees
- CosD - cosine in degrees

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

SinD - sine in degrees
Power - power a^n
Log - log base 10
Exp10 - exp base 10
Sign - negative=-1 positive=1 null=0
IntSign - negative=-1 positive=1 null=0
IntSqrt - integer square root
IntPower - integer power a^n

}

Const

Ln10 = 2.30258509299405E+000; { Ln(10) = 2.3025651 }
OneOverLn10 = 0.43429448190325E+000; { 1 / Ln(10) =
0.4342945 }
PiOver180 = 1.74532925199433E-002; { Pi / 180 =
0.0174533 }
PiUnder180 = 5.72957795130823E+001; { 180 / Pi =
57.2957795 }

Function Radians(Angle : Real) : Real;

Begin

Radians := Angle * PiOver180 { Angle * Pi / 180 }

End;

Function Degrees(Angle : Real) : Real;

Begin

Degrees := Angle * PiUnder180 { Angle * 180 / Pi }

End;

Function CosD(Angle : Real) : Real;

Begin

CosD := Cos(Radians(Angle))

HIGH-RESOLUTION GRAPHICS

End;

Function SinD(Angle : Real) : Real;

Begin

SinD := Sin(Radians(Angle))

End;

Function Power(Base : Real; Exponent : Integer) : Real;

Var

BPower : Real;

t : Integer;

Begin

If (Exponent = 0)

Power := 1

Else

Begin

BPower := 1.0;

For t := 1 To Exponent Do

BPower := BPower * Base;

Power := BPower

End

End;

Function Log(x : Real) : Real;

Begin

Log := Ln(x) * OneOverLn10

End;

Function Exp10(x : Real) : Real;

Begin

Exp10 := Exp(x * Ln10)

End;

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
Function Sign(x : Real) : Integer;
```

```
Begin
  If (x < 0)
    Sign := -1
  Else
    If (x > 0)
      Sign := 1
    Else
      Sign := 0
End;
```

```
Function IntSign(x : Integer) : Integer;
```

```
Begin
  If (x < 0)
    IntSign := -1
  Else
    If (x > 0)
      IntSign := 1
    Else
      IntSign := 0
End;
```

```
Function IntSqrt(x : Integer) : Integer;
```

```
Var
```

```
OddInt, OldArg, FirstSqrt : Integer;
```

```
Begin
```

```
OddInt := 1;
```

```
OldArg := x;
```

```
While x >= 0 Do
```

```
Begin
```

```
  x := x - OddInt;
```

```
  OddInt := OddInt + 2
```

```
End;
```

```
FirstSqrt := OddInt Shr 1;
```

```
If (Sqr(FirstSqrt) - FirstSqrt + 1 > OldArg) { regulate
                                              overshoot }
```



```

    IntSqrt := FirstSqrt - 1
Else
    IntSqrt := FirstSqrt
End;

```

```

Function IntPower(Base, Exponent : Integer) : Integer;

```

```

Begin
    If Exponent = 0
        IntPower := 1
    Else
        IntPower := Base * IntPower(Base, Exponent - 1)
    End;

```

```

{

```

Vector and Matrix Routines

Vec	- Make Vector	
VecInt	- Make Integer Vector	
UnVec	- Get Components of Vector	
UnVecInt	- Get Components of Integer Vector	
VecDot	- Vector Dot Product = $A \cdot B$	
VecCross	- Vector Cross Product	$C = A \times B$
VecLen	- Vector Length = $ A $	
VecNormalize	- Vector Normalize	$B = A / A $
VecMatxMult	- Vector Matrix Multiply	$B = \text{Matx} \times A$
VecSub	- Vector Subtraction	$C = A - B$
VecSubInt	- Vector Subtraction Integer	$C = A - B$
VecAdd	- Vector Addition	$C = A + B$
VecAdd3	- Vector Addition	$D = A + B + C$
VecCopy	- Vector Copy	$B = A$
VecLinComb	- Vector Linear Combination	$C = rA + sB$
VecScalMult	- Vector Scalar Multiple	$B = rA$

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

VecScalMultI - Vector Scalar Multiple $B = rA$, A is integer
VecScalMultInt - Vector Scalar Multiple and Rounding $B = \text{Int}(rA)$
VecAddScalMult - Vector Add Scalar Multiple $C = rA + B$
VecNull - Vector Null $C = 0$
VecNullInt - Vector Null Integer $C = 0$
VecElemMult - Vector Element Multiply $C = r * A * B$

}

Type

TDA = Array[0..2] Of Real;
TDIA = Array[0..2] Of Integer;
FDA = Array[0..3] Of Real;
Matx4x4 = Array[0..3, 0..3] Of Real;

Procedure Vec(r, s, t : Real; Var A : TDA); { Make Vector $A = ri + sj + tk$ }

Begin

A[0] := r;
A[1] := s;
A[2] := t

End;

Procedure VecInt(r, s, t : Integer; Var A : TDIA);
{ Make Integer Vector $A = ri + sj + tk$ }

Begin

A[0] := r;
A[1] := s;
A[2] := t

End;

Procedure UnVec(A : TDA; Var r, s, t : Real);
{ Get Components of Vector }

Begin

r := A[0];
s := A[1];


```

t := A[2]
End;

```

```

Procedure UnVecInt(A : TDIA; Var r, s, t : Integer);
    { Get Components of Integer Vector }
Begin
    r := A[0];
    s := A[1];
    t := A[2]
End;

```

```

Function VecDot(A, B : TDA) : Real; {Vector Dot Product =  $A \cdot B$ }
Begin
    VecDot := A[0]*B[0] + A[1]*B[1] + A[2]*B[2]
End;

```

```

Procedure VecCross(A, B : TDA; Var C : TDA);
    { Vector Cross Product  $C = A \times B$  }
Begin
    C[0] := A[1]*B[2] - A[2]*B[1];
    C[1] := A[2]*B[0] - A[0]*B[2];
    C[2] := A[0]*B[1] - A[1]*B[0]
End;

```

```

Function VecLen(A : TDA) : Real; { Vector Length =  $|A|$  }
Begin
    VecLen := Sqrt( Sqr(A[0]) + Sqr(A[1]) + Sqr(A[2]) )
End;

```

```

Procedure VecNormalize(Var A : TDA); { Vector Normalize  $B = A$ 
    / $|A|$  }
Var
    dist, invdist : Real;

```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
Begin
  dist := VecLen(A);
  If Not(dist = 0.0)
    Begin
      invdist := 1 / dist;
      A[0] := A[0] * invdist;
      A[1] := A[1] * invdist;
      A[2] := A[2] * invdist;
    End
  Else
    Begin
      WriteLn('Zero-Length Vectors cannot be Normalized');
      Halt
    End
  End;
End;
```

```
Procedure VecMatxMult(A : FDA; Matrix : Matx4x4; Var B : FDA);
{ Vector Matrix Multiply B = Matx x A }
```

```
Var
  mRow, mCol : Integer;

Begin
  For mCol := 0 To 3 Do
    Begin
      B[mCol] := 0;
      For mRow := 0 To 3 Do
        B[mCol] := B[mCol] + A[mRow] * Matrix[mRow,mCol]
      End
    End
  End;
End;
```

```
Procedure VecSub(A, B : TDA; Var C : TDA);{ Vector Subtraction
C = A - B }
```

```
Begin
  C[0] := A[0] - B[0];
  C[1] := A[1] - B[1];
  C[2] := A[2] - B[2]
End;
```


HIGH-RESOLUTION GRAPHICS

```
Procedure VecSubInt(A, B : TDIA; Var C : TDA);  
    { Vector Subtraction Integer to Real  $C = A - B$  }
```

```
Begin
```

```
    C[0] := A[0] - B[0];
```

```
    C[1] := A[1] - B[1];
```

```
    C[2] := A[2] - B[2]
```

```
End;
```

```
Procedure VecAdd(A, B : TDA; Var C : TDA); { Vector Addition  $C$   
                                             $= A + B$  }
```

```
Begin
```

```
    C[0] := A[0] + B[0];
```

```
    C[1] := A[1] + B[1];
```

```
    C[2] := A[2] + B[2]
```

```
End;
```

```
Procedure VecAdd3(A, B, C : TDA; Var D : TDA);  
    { Vector Addition  $D = A + B + C$  }
```

```
Begin
```

```
    D[0] := A[0] + B[0] + C[0];
```

```
    D[1] := A[1] + B[1] + C[1];
```

```
    D[2] := A[2] + B[2] + C[2]
```

```
End;
```

```
Procedure VecCopy(A : TDA; Var B : TDA); { Vector Copy  $B = A$  }
```

```
Begin
```

```
    B := A
```

```
End;
```

```
Procedure VecLinComb(r : Real; A : TDA; s : Real; B : TDA;  
    Var C : TDA);  
    { Vector Linear Combination  $C = rA + sB$  }
```

```
Begin
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
    C[0] := r * A[0] + s * B[0];  
    C[1] := r * A[1] + s * B[1];  
    C[2] := r * A[2] + s * B[2]  
End;
```

```
Procedure VecScalMult(r : Real; A : TDA; Var B : TDA);  
    { Vector Scalar Multiple B = rA }
```

```
Begin  
    B[0] := r * A[0];  
    B[1] := r * A[1];  
    B[2] := r * A[2]  
End;
```

```
Procedure VecScalMultI(r : Real; A : TDIA; Var B : TDA);  
    { Vector Scalar Multiple B = rA, A is integer }
```

```
Begin  
    B[0] := r * A[0];  
    B[1] := r * A[1];  
    B[2] := r * A[2]  
End;
```

```
Procedure VecScalMultInt(r : Real; A : TDA; Var B : TDIA);  
    { Vector Scalar Multiple and Rounding B = Int(rA) }
```

```
Begin  
    B[0] := Round(r * A[0]);  
    B[1] := Round(r * A[1]);  
    B[2] := Round(r * A[2])  
End;
```

```
Procedure VecAddScalMult(r : Real; A, B : TDA; Var C : TDA);  
    { Vector Add Scalar Multiple C = rA + B }
```

```
Begin  
    C[0] := r * A[0] + B[0];  
    C[1] := r * A[1] + B[1];  
    C[2] := r * A[2] + B[2]  
End;
```


HIGH-RESOLUTION GRAPHICS

```
Procedure VecNull(Var A : TDA); { Vector Null C = 0 }
```

```
Begin
```

```
  A[0] := 0.0;
```

```
  A[1] := 0.0;
```

```
  A[2] := 0.0
```

```
End;
```

```
Procedure VecNullInt(Var A : TDIA); {Vector Null Integer C = 0}
```

```
Begin
```

```
  A[0] := 0;
```

```
  A[1] := 0;
```

```
  A[2] := 0
```

```
End;
```

```
Procedure VecElemMult(r : Real; A, B : TDA; Var C : TDA);  
  { Vector Element Multiply C = r * A * B }
```

```
Begin
```

```
  C[0] := r * A[0] * B[0];
```

```
  C[1] := r * A[1] * B[1];
```

```
  C[2] := r * A[2] * B[2]
```

```
End;
```

```
{
```

Affine Transformation Routines

ZeroMatrix - zeros the elements of a 4x4 matrix
Translate3D - make translation matrix
Scale3D - make scaling matrix
Rotate3D - make rotation matrix
ZeroAllMatrices - zeros all matrices used in transformation
Multiply3DMatrices - multiply two 4x4 matrices

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

PrepareMatrix - prepare the transformation matrix ($T_m = S \cdot R \cdot T$)
PrepareInvMatrix - prepare the inverse transformation matrix
Transform - multiply a vertex by the transformation matrix

}

Procedure ZeroMatrix(Var A : Matx4x4);

Var
i, j : Integer;

Begin
For i := 0 To 3 Do
For j := 0 To 3 Do
A[i,j] := 0.0
End;

Procedure Translate3D(tx, ty, tz : Real; Var A : Matx4x4);

Var
i : Integer;

Begin
ZeroMatrix(A);
For i := 0 To 3 Do
A[i,i] := 1.0;
A[0,3] := -tx;
A[1,3] := -ty;
A[2,3] := -tz
End;

Procedure Scale3D(sx, sy, sz : Real; Var A : Matx4x4);

Begin
ZeroMatrix(A);
A[0,0] := sx;
A[1,1] := sy;
A[2,2] := sz;

HIGH-RESOLUTION GRAPHICS

```
A[3,3] := 1.0  
End;
```

```
Procedure Rotate3D(m : Integer; Theta : Real; Var A : Matx4x4);
```

```
Var  
  m1, m2 : Integer;  
  c, s : Real;
```

```
Begin  
  ZeroMatrix(A);  
  A[m-1,m-1] := 1.0;  
  A[3,3] := 1.0;  
  m1 := (m Mod 3) + 1;  
  m2 := (m1 Mod 3);  
  m1 := m1 - 1;  
  c := CosD( Theta );  
  s := SinD( Theta );  
  A[m1,m1] := c;  
  A[m1,m2] := s;  
  A[m2,m2] := c;  
  A[m2,m1] := -s
```

```
End;
```

```
Procedure Multiply3DMatrices(A, B : Matx4x4; Var C :  
  Matx4x4);
```

```
Var  
  i, j, k : Integer;  
  ab : Real;
```

```
Begin  
  For i := 0 To 3 Do  
    For j := 0 To 3 Do  
      Begin  
        ab := 0;  
        For k := 0 To 3 Do  
          ab := ab + A[i,k] * B[k,j];
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
        C[i,j] := ab
    End
End;

Procedure PrepareMatrix(Tx, Ty, Tz, Sx, Sy, Sz, Rx, Ry, Rz :
                        Real; Var XForm : Matx4x4);

    Var
        M1, M2, M3, M4, M5, M6, M7, M8, M9 : Matx4x4;

    Begin
        Scale3D ( Sx, Sy, Sz, M1 );
        Rotate3D ( 1, Rx, M2 );
        Rotate3D ( 2, Ry, M3 );
        Rotate3D ( 3, Rz, M4 );
        Translate3D( Tx, Ty, Tz, M5 );
        Multiply3DMatrices( M2, M1, M6 );
        Multiply3DMatrices( M3, M6, M7 );
        Multiply3DMatrices( M4, M7, M8 );
        Multiply3DMatrices( M5, M8, M9 );
        XForm := M9
    End;

Procedure PrepareInvMatrix(Tx, Ty, Tz, Sx, Sy, Sz, Rx, Ry, Rz :
                           Real; Var XForm : Matx4x4);

    Var
        M1, M2, M3, M4, M5, M6, M7, M8, M9 : Matx4x4;

    Begin
        Scale3D ( Sx, Sy, Sz, M1 );
        Rotate3D ( 1, Rx, M2 );
        Rotate3D ( 2, Ry, M3 );
        Rotate3D ( 3, Rz, M4 );
        Translate3D( Tx, Ty, Tz, M5 );
        Multiply3DMatrices( M4, M5, M6 );
        Multiply3DMatrices( M3, M6, M7 );
        Multiply3DMatrices( M2, M7, M8 );
```


HIGH-RESOLUTION GRAPHICS

```
Multiply3DMatrices( M1, M8, M9 );  
XForm := M9  
End;
```

```
Procedure Transform(A : TDA; M : Matx4x4; Var B : TDA);  
  
Begin  
  B[0] := M[0,0] * A[0] + M[0,1] * A[1] + M[0,2] * A[2] +  
                                                M[0,3];  
  B[1] := M[1,0] * A[0] + M[1,1] * A[1] + M[1,2] * A[2] +  
                                                M[1,3];  
  B[2] := M[2,0] * A[0] + M[2,1] * A[1] + M[2,2] * A[2] +  
                                                M[2,3]  
End;
```

Figure 17-1. Listing of *Math2.inc* File

```
{  
  Video Graphics Array Driver  
  
  Plot      - place pixel to screen  
  ClearPalette - clear palette register  
  SetPalette - set palette register  
  InitPalette1 - 64 levels of gray, red, green and blue  
  InitPalette2 - 7 colors with 35 intensities each - use with  
                                                         Pixel  
  
  CyclePalette - cycle through palette  
  Circle      - circle draw routine  
  Draw       - line draw routine  
  InitGraphics - initialize graphics  
  WaitForKey - wait for keypress  
  ExitGraphics - sound and wait for keypress before exiting  
                                                         graphics  
  
  Title     - set up text screen colors  
}
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
Var
  Regs : Registers;

Procedure SetMode(Mode : Byte);

  Begin
    With Regs Do
      Begin
        AH := 0;
        AL := Mode
      End;
    Intr($10, Regs)
  End;
```

```
Const
  MaxXRes = 1024;
  MaxYRes = 768;
  MaxX    = (MaxXRes-1);
  MaxY    = (MaxYRes-1);
```

```
Var
  XRes, YRes : Integer;
  PreCalcY : Array[0..MaxY] Of Integer;
  HighRes   : Boolean;
```

```
Procedure PreCalc;
```

```
  Var
    j : Integer;
```

```
  Begin
    For j := 0 To MaxY Do
      PreCalcY[j] := 0;
    If HighRes
      For j := 0 To 767 Do
        PreCalcY[j] := 1024 * j
      Else
        For j := 0 To 199 Do
```


HIGH-RESOLUTION GRAPHICS

```
        PreCalcY[j] := 320 * j
End;

Procedure Plot(x, y : Word;  Color : Byte);

Const
    Segment = $A000;

Var
    Offset, Page : Word;

Begin
    If Not((X < 0) Or (Y < 0) Or (X >= XRes) Or (Y >= YRes))
        Begin
            If HighRes
                Begin
                    Offset := PreCalcY[y] + x; { calculate address
                                                offset in page }
                    Page := y Div 64;         { select graphics
                                                memory page }

                    Port[$3CD] := Page Or 64;
                    MEM[Segment:Offset] := Color
                End
            Else
                Begin
                    Offset := PreCalcY[y] + x; { calculate address
                                                offset in page }
                    MEM[Segment:Offset] := Color
                End
            End
        End
    End;

Type
    RGB = Record
        Red, Grn, Blu : Byte
    End;
    PaletteRegister = Array[0..255] Of RGB;
Var
    Color : PaletteRegister;
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
Procedure ClearPalette(Var Color : PaletteRegister);
```

```
  Var
```

```
    i : Byte;
```

```
  Begin
```

```
    For i := 0 To 255 Do
```

```
      Begin
```

```
        Color[i].Red := 0;
```

```
        Color[i].Grn := 0;
```

```
        Color[i].Blu := 0
```

```
      End
```

```
  End;
```

```
Procedure SetPalette(Hue : PaletteRegister);
```

```
  Begin
```

```
    With Regs Do
```

```
      Begin
```

```
        AX := $1012;
```

```
        BX := 0;
```

```
        CX := 256;
```

```
        ES := Seg(Hue);
```

```
        DX := OfS(Hue)
```

```
      End;
```

```
      Intr($10, Regs)
```

```
  End;
```

```
Procedure InitPalette1(Var Color : PaletteRegister);
```

```
  Var
```

```
    i : Byte;
```

```
  Begin
```

```
    For i := 0 To 63 Do      { Gray 0-63 }
```

```
      Begin
```

```
        Color[i].Red := i;
```

```
        Color[i].Grn := i;
```


HIGH-RESOLUTION GRAPHICS

```
    Color[i].Blu := i
End;
For i := 64 To 127 Do { Red 0-63 }
Begin
    Color[i].Red := i - 64;
    Color[i].Grn := 0;
    Color[i].Blu := 0
End;
For i := 128 To 191 Do { Green 0-63 }
Begin
    Color[i].Red := 0;
    Color[i].Grn := i - 128;
    Color[i].Blu := 0
End;
For i := 192 To 255 Do { Blue 0-63 }
Begin
    Color[i].Red := 0;
    Color[i].Grn := 0;
    Color[i].Blu := i - 192
End
End;
```

Procedure InitPalette2(Var Color : PaletteRegister);

Var

i : Byte;

Begin

For i := 0 To 35 Do { Blue }

Begin

Color[i].Red := 0;

Color[i].Grn := 0;

Color[i].Blu := Round(1.8*i)

End;

For i := 36 To 71 Do { Green }

Begin

Color[i].Red := 0;

Color[i].Grn := Round(1.8*(i-36));

Color[i].Blu := 0

End;

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
For i := 72 To 107 Do { Cyan }
  Begin
    Color[i].Red := 0;
    Color[i].Grn := Round(1.8*(i-72));
    Color[i].Blu := Round(1.8*(i-72))
  End;
For i := 108 To 143 Do { Red }
  Begin
    Color[i].Red := Round(1.8*(i-108));
    Color[i].Grn := 0;
    Color[i].Blu := 0
  End;
For i := 144 To 179 Do { Magenta }
  Begin
    Color[i].Red := Round(1.8*(i-144));
    Color[i].Grn := 0;
    Color[i].Blu := Round(1.8*(i-144))
  End;
For i := 180 To 215 Do { Brown }
  Begin
    Color[i].Red := Round(1.8*(i-180));
    Color[i].Grn := Round(1.8*(i-180));
    Color[i].Blu := 0
  End;
For i := 216 To 251 Do { Gray }
  Begin
    Color[i].Red := Round(1.8*(i-216));
    Color[i].Grn := Round(1.8*(i-216));
    Color[i].Blu := Round(1.8*(i-216))
  End
End;
```

Procedure CyclePalette(Var Hue : PaletteRegister);

```
Var
  i : Byte;
  Tmp : RGB;
```

```
Begin
```


HIGH-RESOLUTION GRAPHICS

```
    Tmp := Hue[0];
    For i := 1 To 255 Do
        Hue[i-1] := Hue[i];
    Hue[255] := Tmp;
    SetPalette(Hue)
End;
```

```
Procedure Swap(Var first, second : Integer);
```

```
    Var
        temp : Integer;
    Begin
        temp := first;
        first := second;
        second := temp
    End;
```

```
Procedure Circle(x, y, Radius : Word; Color : Byte);
```

```
    Var
        a, af, b, bf, target, r2, asp : Integer;

    Begin
        If HighRes
            asp := 100
        Else
            asp := 120;
        target := 0;
        a := radius;
        b := 0;
        r2 := Sqr(radius);
        While a >= b Do
            Begin
                b := Round(Sqrt(r2 - sqr(a)));
                Swap( target, b );
                While b < target Do
                    Begin { Inner loop for straight lines at
                                cardinal points }
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
        af := (asp * a) Div 100; { aspect correction }
        bf := (asp * b) Div 100;
        Plot(x + af, y + b, color); { 8 points plotted
                                     - symmetry}
        Plot(x + bf, y + a, color);
        Plot(x - af, y + b, color);
        Plot(x - bf, y + a, color);
        Plot(x - af, y - b, color);
        Plot(x - bf, y - a, color);
        Plot(x + af, y - b, color);
        Plot(x + bf, y - a, color);
        b := b + 1
    End;
    a := a - 1
End
End;
```

```
Procedure Draw( xx1, yy1, xx2, yy2 : Word;  color : Byte );

Var
    LgDelta, ShDelta, Cycle, LgStep, ShStep, dtotal : Integer;

Begin
    LgDelta := xx2 - xx1;      { delta and step calculations }
    ShDelta := yy2 - yy1;
    If LgDelta < 0
    Begin
        LgDelta := -LgDelta;
        LgStep := -1
    End
    Else
        LgStep := 1;
    If ShDelta < 0
    Begin
        ShDelta := -ShDelta;
        ShStep := -1
    End
    Else
        ShStep := 1;
```


HIGH-RESOLUTION GRAPHICS

```
If ShDelta < LgDelta
Begin { X-axis longer => Cycle = LargeDelta / 2 and use
                                         steps as-is }
```

```
  Cycle := LgDelta SHR 1;
  While xx1 <> xx2 Do { While not at termination }
```

```
    Begin
      Plot(xx1,yy1,color);
      Cycle := Cycle + ShDelta;
      If Cycle > LgDelta
        Begin
          Cycle := Cycle - LgDelta;
          yy1 := yy1 + ShStep
        End;
      xx1 := xx1 + LgStep
```

```
    End;
    Plot(xx1,yy1,color)
```

```
  End
```

```
Else
```

```
  Begin { X-axis Shorter => Cycle = ShortDelta / 2 and
                                         switch steps }
```

```
    Cycle := ShDelta SHR 1;
    Swap(LgDelta, ShDelta);
    Swap(LgStep, ShStep);
    While yy1 <> yy2 Do { While not at termination }
```

```
      Begin
        Plot(xx1,yy1,color);
        Cycle := Cycle + ShDelta;
        If Cycle > LgDelta
          Begin
            Cycle := Cycle - LgDelta;
            xx1 := xx1 + ShStep
```

```
          End;
          yy1 := yy1 + LgStep
```

```
      End;
```

```
      Plot(xx1,yy1,color)
```

```
    End
```

```
End;
```

```
Procedure InitGraphics(Mode : Byte);
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
Begin
  Case Mode Of
    19 : Begin
      HighRes := False;
      XRes := 320;
      YRes := 200
    End;
    56 : Begin
      HighRes := True;
      XRes := 1024;
      YRes := 768
    End
  End;
  PreCalc;
  SetMode(Mode);
  ClearPalette(Color);
  InitPalette2(Color);
  SetPalette(Color)
End;
```

Procedure WaitForKey;

```
Var
  ch, c : Char;

Begin
  Repeat
    Repeat
      ch := ReadKey
    Until (ch<>#0);
    c := UpCase(ch);
  Until (c > '')
End;
```

Procedure ExitGraphics;

```
Begin
  Sound(1000);
```


HIGH-RESOLUTION GRAPHICS

```
Delay(500);
NoSound;
WaitForKey;
SetMode(3)
End;
```

Procedure Title;

```
Begin
  TextColor(Yellow);
  TextBackground(Blue);
  ClrScr
End;
```

```
{
```

Three-Dimensional Plotting Routines

```
InitPlotting      - rotation and tilt angles
InitPerspective   - observer location and distances
MapCoordinates    - maps 3D space onto the 2D screen
CartesianPlot3D   - plot a cartesian system point
CylindricalPlot3D - plot a cylindrical system point
SphericalPlot3D   - plot a spherical system point
DrawLine3D        - plots a line from 3D coordinates
```

```
}
```

Var

```
CentreX, CentreY : Integer;
Angl, Tilt        : Integer;
CosA, SinA        : Real;
CosB, SinB        : Real;
CosACosB, SinASinB : Real;
CosASinB, SinACosB : Real;
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
Procedure InitPlotting(Ang, Tlt : Integer);
```

```
Begin
```

```
  CentreX := XRes Div 2;
```

```
  CentreY := YRes Div 2;
```

```
  Angl := Ang;
```

```
  Tilt := Tlt;
```

```
  WriteLn('View Direction is ', Angl:4, '° around the z-Axis  
                                                and');
```

```
  WriteLn('                                ', Tilt:4, '° off of the z-Axis');
```

```
  CosA := CosD( Angl );
```

```
  SinA := SinD( Angl );
```

```
  CosB := CosD( Tilt );
```

```
  SinB := SinD( Tilt );
```

```
  CosACosB := CosA * CosB;
```

```
  SinASinB := SinA * SinB;
```

```
  CosASinB := CosA * SinB;
```

```
  SinACosB := SinA * CosB
```

```
End;
```

```
Var
```

```
  PerspectivePlot : Boolean;
```

```
  Mx, My, Mz, ds : Real;
```

```
Procedure InitPerspective(Perspective : Boolean; x, y, z, m :  
                                                                Real);
```

```
Begin
```

```
  PerspectivePlot := Perspective;
```

```
  Mx := x;
```

```
  My := y;
```

```
  Mz := z;
```

```
  ds := m
```

```
End;
```

```
Procedure MapCoordinates(X, Y, Z : Real; Var Xp, Yp : Integer);
```



```

Var
  Xt, Yt, Zt : Real;
  OneOverZt   : Real;

Begin
  Xt := (Mx + X * CosA - Y * SinA);
  Yt := (My + X * SinASinB + Y * CosASinB + Z * CosB);
  If PerspectivePlot
    Begin
      Zt := Mz + X * SinACosB + Y * CosACosB - Z * SinB;
      OneOverZt := 1 / Zt;
      Xp := CentreX + Round( ds * Xt * OneOverZt );
      Yp := CentreY - Round( ds * Yt * OneOverZt );
    End
  Else
    Begin
      Xp := CentreX + Round( Xt );
      Yp := CentreY - Round( Yt );
    End
  End;

```

```

Procedure CartesianPlot3D(X, Y, Z : Real; Color : Byte);

```

```

Var
  Xp, Yp : Integer;

Begin
  MapCoordinates(X, Y, Z, Xp, Yp);
  Plot( Xp, Yp, Color );
End;

```

```

Procedure CylindricalPlot3D(Rho, Theta, Z : Real; Color : Byte);

```

```

Var
  X, Y : Real;

Begin
  Theta := Radians( Theta );

```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
    X := Rho * Cos( Theta );
    Y := Rho * Sin( Theta );
    CartesianPlot3D( X, Y, Z, Color )
End;
```

```
Procedure SphericalPlot3D(R, Theta, Phi : Real; Color : Byte);
```

```
    Var
        X, Y, Z : Real;

    Begin
        Theta := Radians( Theta );
        Phi    := Radians( Phi );
        X := R * Sin( Theta ) * Cos( Phi );
        Y := R * Sin( Theta ) * Sin( Phi );
        Z := R * Cos( Theta );
        CartesianPlot3D( X, Y, Z, Color )
    End;
```

```
Procedure DrawLine3D(Pnt1, Pnt2 : TDA; Color : Byte);
```

```
    Var
        Xp1, Yp1 : Integer;
        Xp2, Yp2 : Integer;
        x1, y1, z1 : Real;
        x2, y2, z2 : Real;

    Begin
        UnVec(Pnt1, x1, y1, z1);
        UnVec(Pnt2, x2, y2, z2);
        MapCoordinates(x1, y1, z1, Xp1, Yp1);
        MapCoordinates(x2, y2, z2, Xp2, Yp2);
        Draw(Xp1, Yp1, Xp2, Yp2, Color)
    End;
```


{

Pixel

PutPixel - plots pixel

GetPixel - gets pixel

}

{ Color 1 - Blue }

{ 2 - Green }

{ 3 - Cyan }

{ 4 - Red }

{ 5 - Magenta }

{ 6 - Brown/Yellow }

{ 7 - Gray Scale }

{ Intensity levels (0..35) for each color }

Const

MaxCol = 7;

MaxInten = 35;

Procedure PutPixel(x, y : Integer; Color, Intensity : Byte);

Var

Col : Byte;

Begin

If Intensity > MaxInten Halt;

Col := (MaxInten + 1) * (Color-1) + Intensity;

Plot(x, y, Col)

End;

Function GetPixel(x, y : Integer) : Integer;

Begin

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
    Regs.AX := 3328;  
    Regs.DX := y;  
    Regs.CX := x;  
    Intr($10, Regs);  
    GetPixel := Regs.AX And 255  
End;
```

```
{
```

Setup of Coordinate Axes and Color Palette
--

```
PutAxisAndPalette - toggle for Axis and Palette  
AxisAndPalette - places Axis and Color Palette on screen
```

```
}
```

```
Var
```

```
    DrawAxisAndPalette : Boolean;
```

```
Procedure PutAxisAndPalette(PlaceOnScreen : Boolean);
```

```
    Begin
```

```
        If PlaceOnScreen
```

```
            DrawAxisAndPalette := True
```

```
        Else
```

```
            DrawAxisAndPalette := False
```

```
    End;
```

```
Procedure AxisAndPalette;
```

```
    Procedure DisplayAxis;
```

```
        Var
```

```
            x, y, z : Integer;
```

```
        Begin
```


HIGH-RESOLUTION GRAPHICS

```
For x := -100 To 100 Do CartesianPlot3D(x, 0, 0, 35);
                                { Blue }
CartesianPlot3D(100, 0, 0, 251);    { White }
For y := -100 To 100 Do CartesianPlot3D(0, y, 0, 71);
                                {Green}
CartesianPlot3D(0, 100, 0, 251);    { White }
For z := -100 To 100 Do CartesianPlot3D(0, 0, z, 107);
                                {Cyan}
CartesianPlot3D(0, 0, 100, 251)    { White }
End;
```

Procedure DisplayPalette;

Var

 X, Y : Integer;

 Color : Byte;

 Intensity : Byte;

Begin

 For Color := 1 To MaxCol Do

 For Intensity := 0 To MaxInten Do

 For X := 0 To 3 Do

 For Y := 0 To 3 Do

 PutPixel(X + 5*Color, 190 - Y - 5*Intensity,

 Color, Intensity)

 End

 End

 End

 End

End;

Begin

 If DrawAxisAndPalette

 Begin

 DisplayAxis;

 DisplayPalette

 End

 End;

Figure 17-2. Listing of the *Graph2.inc* File

Defining a Scene: The .RT File

Before you can begin ray-tracing, you need to define the scene. This is done by means of a file ending with the extension *.RT*, which is a text file you can create on your standard text editor, as long as it can output a plain ASCII text file (i.e., a data format that doesn't include any control-characters or binary data). However, the order in which you specify things, and what should appear on each line are constrained. Figure 18-1 lists all the *.RT* files created for scenes thus far. Once you use these scenes and understand how they are created, you can create any scenes imaginable made up of the primitive objects defined in the appropriate files. What follows is a description of the make-up of a typical *.RT* file in the section that follows. Chapter 19 describes how the ray-tracing program uses this file, which will give you a better idea of the constraints involved.

Formatting Constraints

The files listed in this chapter are all text files and are read into the ray-tracing program using Turbo Pascal's *ReadLn* function. This facilitates creating and processing the files, but it requires that the arrangement of elements and positioning of elements on a line be rigidly set. The categories you see flush with the left-hand margin are object definitions occurring in the file in any order. However, once you specify one of these objects, certain numerical values and/or names must occur at the exact positions where they are shown. Just the first file, called *Example1.RT* will be discussed in-depth. However, the problems and reasoning are similar for all the files. The first parameter specified is *STATS*. Note that the following five lines specify the parameters *OUTFILE*, *XRES*, *YRES*, *XPITCH*, and *YPITCH*. The program reads this

file and takes the string (of no more than 30 characters), beginning at column 14 of the first line below *STATS*, and sets it up as the name of the output file. Similarly, the eight characters beginning at column 14 of the next line will be the integer value for *x* resolution; eight characters at column 14 on the next line will be the integer value of *y* resolution, eight characters at column 14 on the next line will be the floating-point value for the *x* pitch (a scaler which is like the inverse of the focal length), and the eight characters beginning at column 14 on the next line will be the floating-point value for the *y* pitch. It doesn't make any difference what you call these. If you made the first line *XRES* the output would still be the name for the output file. So don't get confused as to where things should go; use one of the sample files as a template. Similarly, be sure to start your inputs at column 14. If you don't, the numbers stored by the program may be different from what you think.

The next category listed is *ENVIRONMENT*. There are four weighting factors to determine when a ray is no longer making a significant contribution to the overall light, plus a depth factor that indicates how many recursions should take place to continue tracing the ray before it is no longer significant. You'll note that all of the sample files use the same values for these parameters. Until you have studied the ray-tracing program very carefully and have a good understanding of how these factors interact, leave them alone.

The next category is *OBSERVER*. The first parameter is the focal length. Unfortunately, this doesn't relate directly to any similar values for lenses. However, the principle is the same. Large focal lengths give telephoto effects, with a narrow field of view, and distance compression. Small focal lengths give large fields of view (the wide-angle effect) and considerable distortion. The value of 1.0 used in most of the example files is too wide of an angle. A value of about 3.2 seems to be good. However, note that as you change the focal length, the coverage changes, so you may have to move the observer position to compensate. The following parameters are the observer position vector, the observer rotation angle, and the observer tilt angle. These all have a substantial effect upon the appearance of the scene.

The next category is *SKY*. It has two vector parameters, defining the color of the sky at the horizon and at the zenith. If your scene doesn't include any sky, you won't need to worry about this one.

The next category is *AMBIENTLIGHT*. It has a single parameter, the ambient-light-intensity color vector. Ambient light is not a major contributor, so you probably don't need to worry about it too much, either.

The next category is *LAMP*. You need one of these for each light source (including the sun or moon if you're outdoors). Its parameters are the vector marking the position of the light source, the radius of the light source, and the color vectors indicating the red, green, and blue components of the light source.

The next category is *MATERIAL*. You'll need one of these for every type of material that you define. Its first parameter is a material type name, to which you can assign whatever you wish. The second parameter is the name of a texture. This may be any of the following existing textures, *SMOOTH*, *CHECKER*, *GRIT*, *MARBLE*, *WOOD*, and *SHEETROCK*. The texture name is followed by three color vectors defining the colors of the ambient, diffuse and spectral reflections from the material. This is followed by the glossiness factor, which is the power used in the Phong shading model.

The remainder of each file is dedicated to the definition of the primitive objects which make up the scene. For the *SPHERE*, for example, all that is needed is the location of the center of the sphere, its radius, and the material. Looking at the sample files, you'll see what parameters define each of the other primitive objects.

Example 1
By Christopher D. Watkins

STATS

```
OUTFILE = Example1.CPR
XRES    = 320
YRES    = 200
XPITCH  = 1.000
YPITCH  = 1.000
```

ENVIRONMENT

```
LOCLWGT = 0.700 0.700 0.700
REFLWGT = 0.300 0.300 0.300
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

MINWGT	=	0.030	0.030	0.030
MAXWGT	=	1.000	1.000	1.000
RDEPTH	=	5		

OBSERVER

FLENGTH	=	1.000		
OBSPOS	=	-30.000	-140.000	80.000
ROTATE	=	20.000		
TILT	=	20.000		

SKY

HORCOL	=	0.000	0.000	0.000
ZENCOL	=	0.000	0.000	0.000

AMBIENTLIGHT

INTENS	=	0.150	0.150	0.150
--------	---	-------	-------	-------

LAMP

LOC	=	90.000	-40.000	50.000
RADIUS	=	25.000		
INTENS	=	0.000	1.000	0.000

LAMP

LOC	=	-90.000	-40.000	50.000
RADIUS	=	25.000		
INTENS	=	1.000	0.000	0.000

LAMP

LOC	=	0.000	0.000	130.000
RADIUS	=	25.000		
INTENS	=	0.000	0.000	1.000

MATERIAL

TYPE	=	CHROME
TEXTURE	=	SMOOTH

DEFINING A SCENE: THE .RT FILE

```
AMBRFL = 0.000 0.000 0.000
DIFRFL = 0.250 0.250 0.250
SPCRFL = 0.750 0.750 0.750
GLOSS = 40.000
```

MATERIAL

```
TYPE = PLASTICTILE
```

```
TEXTURE = CHECKER
```

```
AMBRFL = 0.100 0.100 0.100
```

```
DIFRFL = 0.700 0.700 0.700
```

```
SPCRFL = 0.200 0.200 0.200
```

```
GLOSS = 4.000
```

CHECK

```
TILE1 = 1.000 0.000 0.000
```

```
TILE2 = 0.200 0.200 0.200
```

```
TILE = 0.02
```

GROUND

```
MATL = PLASTICTILE
```

SPHERE

```
LOC = 0.000 0.000 50.000
```

```
RADIUS = 50.000
```

```
MATL = CHROME
```

Example 2

By Christopher D. Watkins

STATS

```
OUTFILE = Example2.CPR
```

```
XRES = 320
```

```
YRES = 200
```

```
XPITCH = 1.000
```

```
YPITCH = 1.000
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

ENVIRONMENT

LOCLWGT	=	0.700	0.700	0.700
REFLWGT	=	0.300	0.300	0.300
MINWGT	=	0.030	0.030	0.030
MAXWGT	=	1.000	1.000	1.000
RDEPTH	=	5		

OBSERVER

FLENGTH	=	1.000		
OBSPOS	=	-30.000	-160.000	80.000
ROTATE	=	20.000		
TILT	=	20.000		

SKY

HORCOL	=	0.700	0.800	1.000
ZENCOL	=	0.600	0.600	0.700

AMBIENTLIGHT

INTENS	=	0.150	0.150	0.150
--------	---	-------	-------	-------

LAMP

LOC	=	0.000	-20.000	50.000
RADIUS	=	30.000		
INTENS	=	1.000	0.000	0.000

LAMP

LOC	=	100.000	100.000	150.000
RADIUS	=	40.000		
INTENS	=	0.000	1.000	0.000

LAMP

LOC	=	100.000	-165.000	30.000
RADIUS	=	20.000		
INTENS	=	0.000	0.000	1.000

DEFINING A SCENE: THE .RT FILE

MATERIAL

TYPE	=	CHROME		
TEXTURE	=	SMOOTH		
AMBRFL	=	0.000	0.000	0.000
DIFRFL	=	0.250	0.250	0.250
SPCRFL	=	0.750	0.750	0.750
GLOSS	=	40.000		

MATERIAL

TYPE	=	PLASTICTILE		
TEXTURE	=	CHECKER		
AMBRFL	=	0.100	0.100	0.100
DIFRFL	=	0.700	0.700	0.700
SPCRFL	=	0.200	0.200	0.200
GLOSS	=	4.000		

CHECK

TILE1	=	1.000	0.000	0.000
TILE2	=	0.200	0.200	0.200
TILE	=	0.02		

GROUND

MATL	=	PLASTICTILE
------	---	-------------

CYLINDER

LOC	=	-60.000	120.000	50.000
DIRECT	=	1.000	0.000	0.000
RADIUS	=	50.000		
HEIGHT	=	300.000		
MATL	=	CHROME		

CYLINDER

LOC	=	-100.000	100.000	0.000
DIRECT	=	0.000	0.000	1.000
RADIUS	=	35.000		
HEIGHT	=	140.000		
MATL	=	CHROME		

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

CYLINDER

LOC = 100.000-100.000 0.000
DIRECT = 0.000 0.000 1.000
RADIUS = 35.000
HEIGHT = 140.000
MATL = CHROME

CYLINDER

LOC = -100.000 -100.000 30.000
DIRECT = 0.000 1.000 0.000
RADIUS = 30.000
HEIGHT = 100.000
MATL = CHROME

CONE

LOC = 80.00050.000 110.000
DIRECT = 0.000 0.000 1.000
RADIUS = 30.000
HEIGHT = 110.000
MATL = CHROME

Marbled Marbles on Marble
By Christopher D. Watkins

STATS

OUTFILE = Marbles.CPR
XRES = 320
YRES = 200
XPITCH = 1.000
YPITCH = 1.000

ENVIRONMENT

LOCLWGT = 0.800 0.800 0.800
REFLWGT = 0.200 0.200 0.200
MINWGT = 0.030 0.030 0.030
MAXWGT = 1.000 1.000 1.000
RDEPTH = 5

DEFINING A SCENE: THE .RT FILE

OBSERVER

```

FLENGTH = 1.000
OBSPOS = 40.000 -150.000 100.000
ROTATE = 10.000
TILT = 20.000

```

SKY

```

HORCOL = 0.600 0.700 1.000
ZENCOL = 0.500 0.500 0.500

```

AMBIENTLIGHT

```

INTENS = 0.100 0.100 0.100

```

MATERIAL

```

TYPE = ITALIANMARBLE
TEXTURE = MARBLE
AMBRFL = 0.100 0.100 0.100
DIFRFL = 0.700 0.700 0.700
SPCRFL = 0.200 0.200 0.200
GLOSS = 10.000

```

GROUND

```

MATL = ITALIANMARBLE

```

SPHERE

```

LOC = 0.000 0.000 50.000
RADIUS = 50.000
MATL = ITALIANMARBLE

```

SPHERE

```

LOC = 140.000 0.000 50.000
RADIUS = 50.000
MATL = ITALIANMARBLE

```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

SPHERE

LOC = 70.000 150.000 50.000
RADIUS = 50.000
MATL = ITALIANMARBLE

{ sun }

LAMP

LOC = 1000.000 -999.000 1000.000
RADIUS = 700.000
INTENS = 1.000 1.000 0.900

Office Scene
By Christopher D. Watkins

STATS

OUTFILE = Office.CPR
XRES = 320
YRES = 200
XPITCH = 1.000
YPITCH = 1.000

ENVIRONMENT

LOCLWGT = 0.750 0.750 0.750
REFLWGT = 0.250 0.250 0.250
MINWGT = 0.030 0.030 0.030
MAXWGT = 1.000 1.000 1.000
RDEPTH = 5

OBSERVER

FLENGTH = 0.750
OBSPOS = 0.000 -280.000 140.000
ROTATE = 0.000
TILT = 10.000

DEFINING A SCENE: THE .RT FILE

SKY

HORCOL	=	0.000	0.000	0.000
ZENCOL	=	0.000	0.000	0.000

AMBIENTLIGHT

INTENS	=	0.200	0.200	0.200
--------	---	-------	-------	-------

MATERIAL

TYPE	=	MATTEWALL		
TEXTURE	=	SHEETROCK		
AMBRFL	=	0.500	0.500	0.500
DIFRFL	=	0.500	0.500	0.500
SPCRFL	=	0.000	0.000	0.000
GLOSS	=	0.000		

MATERIAL

TYPE	=	CHROME		
TEXTURE	=	SMOOTH		
AMBRFL	=	0.100	0.100	0.100
DIFRFL	=	0.100	0.100	0.100
SPCRFL	=	0.800	0.800	0.800
GLOSS	=	40.000		

MATERIAL

TYPE	=	BRASS		
TEXTURE	=	SMOOTH		
AMBRFL	=	0.100	0.100	0.100
DIFRFL	=	0.300	0.300	0.300
SPCRFL	=	0.600	0.500	0.250
GLOSS	=	30.000		

MATERIAL

TYPE	=	MIRROR		
TEXTURE	=	SMOOTH		
AMBRFL	=	0.100	0.100	0.100
DIFRFL	=	0.200	0.200	0.200

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
SPCRFL  =    0.700    0.700    0.700
GLOSS   =    30.000
```

MATERIAL

```
TYPE    = ITALIANMARBLE
```

```
TEXTURE = MARBLE
```

```
AMBRFL  =    0.200    0.200    0.200
```

```
DIFRFL  =    0.600    0.600    0.600
```

```
SPCRFL  =    0.200    0.200    0.200
```

```
GLOSS   =    10.000
```

MATERIAL

```
TYPE    = OAKWOOD
```

```
TEXTURE = WOOD
```

```
AMBRFL  =    0.200    0.200    0.200
```

```
DIFRFL  =    0.700    0.400    0.300
```

```
SPCRFL  =    0.100    0.100    0.100
```

```
GLOSS   =    5.000
```

```
{ floor }
```

GROUND

```
MATL    = OAKWOOD
```

```
{ end wall }
```

PARALLELOGRAM

```
LOC      = -300.000    300.000    0.000
```

```
V1       =    600.000    0.000    0.000
```

```
V2       =    0.000    0.000    400.000
```

```
MATL     = MATTEWALL
```

```
{ left wall }
```

PARALLELOGRAM

```
LOC      = -300.000   -300.000    0.000
```

```
V1       =    0.000    600.000    0.000
```


DEFINING A SCENE: THE .RT FILE

```
V2      =      0.000      0.000      400.000
MATL    = MATTEWALL
```

```
{ right wall }
```

```
PARALLELOGRAM
```

```
LOC      =      300.000      300.000      0.000
V1       =      0.000     -600.000      0.000
V2       =      0.000      0.000      400.000
MATL     = MATTEWALL
```

```
{ ceiling }
```

```
PARALLELOGRAM
```

```
LOC      =     -300.000      300.000      400.000
V1       =      600.000      0.000      0.000
V2       =      0.000     -600.000      0.000
MATL     = MATTEWALL
```

```
{ back wall }
```

```
PARALLELOGRAM
```

```
LOC      =      300.000     -300.000      400.000
V1       =     -600.000      0.000      0.000
V2       =      0.000      600.000      0.000
MATL     = MATTEWALL
```

```
{ table }
```

```
BOX
```

```
LOC      =     -150.000     -100.000      80.000
V1       =      300.000      0.000      0.000
V2       =      0.000      200.000      0.000
V3       =      0.000      0.000      20.000
MATL     = ITALIANMARBLE
```

```
{ table stand }
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

BOX

LOC	=	-125.000	-75.000	0.000
V1	=	250.000	0.000	0.000
V2	=	0.000	150.000	0.000
V3	=	0.000	0.000	80.000
MATL	=	ITALIANMARBLE		

{ lamp on table }

CIRCLE

LOC	=	125.000	75.000	100.000
V1	=	1.000	0.000	0.000
V2	=	0.000	1.000	0.000
RADIUS	=	25.000		
MATL	=	CHROME		

CYLINDER

LOC	=	125.000	75.000	101.000
DIRECT	=	0.000	0.000	1.000
RADIUS	=	15.000		
HEIGHT	=	18.000		
MATL	=	CHROME		

LAMP

LOC	=	125.000	75.000	145.000
RADIUS	=	25.000		
INTENS	=	1.000	1.000	1.000

{ orb on table }

SPHERE

LOC	=	-125.000	75.000	120.000
RADIUS	=	20.000		
MATL	=	BRASS		

{ pyramid on table }

DEFINING A SCENE: THE .RT FILE

PYRAMID

LOC	=	-125.000	-70.000	100.000
V1	=	20.000	0.000	0.000
V2	=	0.000	20.000	0.000
HEIGHT	=	40.000		
MATL	=	CHROME		

{ box on table }

BOX

LOC	=	120.000	-70.000	100.000
V1	=	20.000	0.000	0.000
V2	=	0.000	20.000	0.000
V3	=	0.000	0.000	20.000
MATL	=	MIRROR		

{ mirror on wall }

PARALLELOGRAM

LOC	=	-150.000	299.000	125.000
V1	=	300.000	0.000	0.000
V2	=	0.000	0.000	150.000
MATL	=	MIRROR		

{ lamp on floor near rear wall }

CIRCLE

LOC	=	-280.000	-280.000	0.000
V1	=	1.000	0.000	0.000
V2	=	0.000	1.000	0.000
RADIUS	=	15.000		
MATL	=	CHROME		

CYLINDER

LOC	=	-280.000	-280.000	1.000
DIRECT	=	0.000	0.000	1.000
RADIUS	=	8.000		

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

HEIGHT = 18.000
MATL = CHROME

LAMP

LOC = -280.000 -280.000 45.000
RADIUS = 15.000
INTENS = 1.000 1.000 1.000

Lakeside Park
By Christopher D. Watkins

STATS

OUTFILE = Park.CPR
XRES = 320
YRES = 200
XPITCH = 1.000
YPITCH = 1.000

ENVIRONMENT

LOCLWGT = 0.700 0.700 0.700
REFLWGT = 0.300 0.300 0.300
MINWGT = 0.030 0.030 0.030
MAXWGT = 1.000 1.000 1.000
RDEPTH = 5

OBSERVER

FLENGTH = 1.000
OBSPOS = -30.000 -180.000 100.000
ROTATE = 20.000
TILT = 23.000

SKY

HORCOL = 0.600 0.700 1.000
ZENCOL = 0.500 0.500 0.500

DEFINING A SCENE: THE .RT FILE

AMBIENTLIGHT

INTENS = 0.150 0.150 0.150

MATERIAL

TYPE = GRASS

TEXTURE = GRIT

AMBRFL = 0.200 0.200 0.200

DIFRFL = 0.600 0.800 0.250

SPCRFL = 0.000 0.000 0.000

GLOSS = 0.000

MATERIAL

TYPE = CEMENT

TEXTURE = GRIT

AMBRFL = 0.200 0.200 0.200

DIFRFL = 0.800 0.800 0.800

SPCRFL = 0.000 0.000 0.000

GLOSS = 0.000

MATERIAL

TYPE = LIGHTCEMENT

TEXTURE = SHEETROCK

AMBRFL = 0.100 0.100 0.100

DIFRFL = 0.900 0.900 0.900

SPCRFL = 0.000 0.000 0.000

GLOSS = 0.000

MATERIAL

TYPE = WATER

TEXTURE = GRIT

AMBRFL = 0.100 0.100 0.300

DIFRFL = 0.100 0.300 0.300

SPCRFL = 0.100 0.100 0.400

GLOSS = 3.000

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

MATERIAL

```
TYPE      = LAKEWATER
TEXTURE   = GRIT
AMBRFL    =      0.100      0.100      0.200
DIFRFL    =      0.100      0.200      0.200
SPCRFL    =      0.100      0.100      0.300
GLOSS     =      2.000
```

MATERIAL

```
TYPE      = BLACKANODIZED
TEXTURE   = SMOOTH
AMBRFL    =      1.000      1.000      1.000
DIFRFL    =      0.000      0.000      0.000
SPCRFL    =      0.000      0.000      0.000
GLOSS     =      0.000
```

MATERIAL

```
TYPE      = MILKWHITEGLASS
TEXTURE   = SMOOTH
AMBRFL    =      0.500      0.500      0.500
DIFRFL    =      0.400      0.400      0.400
SPCRFL    =      0.100      0.100      0.100
GLOSS     =      4.000
```

MATERIAL

```
TYPE      = MIRROR
TEXTURE   = SMOOTH
AMBRFL    =      0.100      0.100      0.100
DIFRFL    =      0.200      0.200      0.200
SPCRFL    =      0.700      0.700      0.700
GLOSS     =     30.000
```

GROUND

```
MATL  = GRASS
```

```
{ sun }
```


DEFINING A SCENE: THE .RT FILE

LAMP

LOC = 1000.000 -999.000 1000.000
 RADIUS = 700.000
 INTENS = 1.000 1.000 0.900

{ birdbath }

CYLINDER

LOC = 0.000 0.000 0.000
 DIRECT = 0.000 0.000 1.000
 RADIUS = 80.000
 HEIGHT = 25.000
 MATL = CEMENT

RING

LOC = 0.000 0.000 25.000
 V1 = 1.000 0.000 0.000
 V2 = 0.000 1.000 0.000
 RAD1 = 70.000
 RAD2 = 80.000
 MATL = CEMENT

CYLINDER

LOC = 0.000 0.000 25.000
 DIRECT = 0.000 0.000 1.000
 RADIUS = 70.000
 HEIGHT = 20.000
 MATL = CEMENT

RING

LOC = 0.000 0.000 45.000
 V1 = 1.000 0.000 0.000
 V2 = 0.000 1.000 0.000
 RAD1 = 65.000
 RAD2 = 70.000
 MATL = CEMENT

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

CIRCLE

LOC	=	0.000	0.000	44.500
V1	=	1.000	0.000	0.000
V2	=	0.000	1.000	0.000
RADIUS	=	68.000		
MATL	=	WATER		

CYLINDER

LOC	=	0.000	0.000	45.000
DIRECT	=	0.000	0.000	1.000
RADIUS	=	6.000		
HEIGHT	=	28.000		
MATL	=	CEMENT		

CONE

LOC	=	0.000	0.000	60.000
DIRECT	=	0.000	0.000	-1.000
RADIUS	=	40.000		
HEIGHT	=	15.000		
MATL	=	CEMENT		

CIRCLE

LOC	=	0.000	0.000	74.000
V1	=	1.000	0.000	0.000
V2	=	0.000	1.000	0.000
RADIUS	=	39.000		
MATL	=	WATER		

{ lake water }

PARALLELOGRAM

LOC	=	-999.000	750.000	1.000
V1	=	9999.000	0.000	0.000
V2	=	0.000	5000.000	0.000
MATL	=	LAKEWATER		

DEFINING A SCENE: THE .RT FILE

```
{ sidewalk }
```

```
PARALLELOGRAM
```

```
LOC      =  -299.000    160.000    1.500
V1       =  1999.000    0.000    0.000
V2       =    0.000    80.000    0.000
MATL     =  LIGHTCEMENT
```

```
{ streetlamps }
```

```
CYLINDER
```

```
LOC      =  -200.000    250.000    0.000
DIRECT   =    0.000    0.000    1.000
RADIUS   =    6.000
HEIGHT   =   100.000
MATL     =  BLACKANODIZED
```

```
SPHERE
```

```
LOC      =  -200.000    250.000    115.000
RADIUS   =   15.000
MATL     =  MILKWHITEGLASS
```

```
CYLINDER
```

```
LOC      =   100.000    250.000    0.000
DIRECT   =    0.000    0.000    1.000
RADIUS   =    6.000
HEIGHT   =   100.000
MATL     =  BLACKANODIZED
```

```
SPHERE
```

```
LOC      =   100.000    250.000    115.000
RADIUS   =   15.000
MATL     =  MILKWHITEGLASS
```

```
CYLINDER
```

```
LOC      =   400.000    250.000    0.000
DIRECT   =    0.000    0.000    1.000
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

RADIUS = 6.000
 HEIGHT = 100.000
 MATL = BLACKANODIZED

SPHERE
 LOC = 400.000 250.000 115.000
 RADIUS = 15.000
 MATL = MILKWHITEGLASS

CYLINDER
 LOC = 700.000 250.000 0.000
 DIRECT = 0.000 0.000 1.000
 RADIUS = 6.000
 HEIGHT = 100.000
 MATL = BLACKANODIZED

SPHERE
 LOC = 700.000 250.000 115.000
 RADIUS = 15.000
 MATL = MILKWHITEGLASS

{ park bench }

CYLINDER
 LOC = 400.000 120.000 0.000
 DIRECT = 0.000 0.000 1.000
 RADIUS = 10.000
 HEIGHT = 35.000
 MATL = CEMENT

CYLINDER
 LOC = 500.000 120.000 0.000
 DIRECT = 0.000 0.000 1.000
 RADIUS = 10.000
 HEIGHT = 35.000
 MATL = CEMENT

BOX
 LOC = 380.000 90.000 35.000

DEFINING A SCENE: THE .RT FILE

```
V1      =    140.000      0.000      0.000
V2      =      0.000      60.000      0.000
V3      =      0.000      0.000      5.000
MATL    = CEMENT
```

{ garbage can }

```
CYLINDER
LOC      =    200.000      120.000      0.000
DIRECT   =      0.000      0.000      1.000
RADIUS   =     16.000
HEIGHT   =     35.000
MATL     = CEMENT
```

{ city }

```
PYRAMID
LOC      =   -200.000      9999.000      0.000
V1       =    400.000      0.000      0.000
V2       =      0.000      400.000      0.000
HEIGHT   =   1200.000
MATL     = CHROME
```

```
BOX
LOC      =    790.000      9999.000      0.000
V1       =    450.000      0.000      0.000
V2       =      0.000      450.000      0.000
V3       =      0.000      0.000      2200.000
MATL     = CEMENT
```

```
CYLINDER
LOC      =   4500.000      9999.000      0.000
DIRECT   =      0.000      0.000      1.000
RADIUS   =    510.000
HEIGHT   =   1500.000
MATL     = MIRROR
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

Lakeside Park at Night
By Christopher D. Watkins

STATS

OUTFILE = ParkNite.CPR
XRES = 320
YRES = 200
XPITCH = 1.000
YPITCH = 1.000

ENVIRONMENT

LOCLWGT = 0.700 0.700 0.700
REFLWGT = 0.300 0.300 0.300
MINWGT = 0.030 0.030 0.030
MAXWGT = 1.000 1.000 1.000
RDEPTH = 5

OBSERVER

FLENGTH = 1.000
OBSPOS = -30.000 -180.000 100.000
ROTATE = 20.000
TILT = 23.000

SKY

HORCOL = 0.050 0.050 0.030
ZENCOL = 0.050 0.050 0.030

AMBIENTLIGHT

INTENS = 0.150 0.150 0.150

MATERIAL

TYPE = GRASS

DEFINING A SCENE: THE .RT FILE

TEXTURE = GRIT

AMBRFL	=	0.200	0.200	0.200
DIFRFL	=	0.600	0.800	0.250
SPCRFL	=	0.000	0.000	0.000
GLOSS	=	0.000		

MATERIAL

TYPE = DULLCEMENT

TEXTURE = GRIT

AMBRFL	=	0.200	0.200	0.200
DIFRFL	=	0.300	0.300	0.300
SPCRFL	=	0.000	0.000	0.000
GLOSS	=	0.000		

MATERIAL

TYPE = CEMENT

TEXTURE = GRIT

AMBRFL	=	0.200	0.200	0.200
DIFRFL	=	0.800	0.800	0.800
SPCRFL	=	0.000	0.000	0.000
GLOSS	=	0.000		

MATERIAL

TYPE = LIGHTCEMENT

TEXTURE = SHEETROCK

AMBRFL	=	0.100	0.100	0.100
DIFRFL	=	0.900	0.900	0.900
SPCRFL	=	0.000	0.000	0.000
GLOSS	=	0.000		

MATERIAL

TYPE = WATER

TEXTURE = GRIT

AMBRFL	=	0.100	0.100	0.300
DIFRFL	=	0.100	0.300	0.300
SPCRFL	=	0.100	0.100	0.400
GLOSS	=	3.000		

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

MATERIAL

```

    TYPE      = LAKEWATER
    TEXTURE   = GRIT
    AMBRFL    =      0.100      0.100      0.200
    DIFRFL    =      0.100      0.200      0.200
    SPCRFL    =      0.100      0.100      0.300
    GLOSS     =      2.000
  
```

MATERIAL

```

    TYPE      = BLACKANODIZED
    TEXTURE   = SMOOTH
    AMBRFL    =      1.000      1.000      1.000
    DIFRFL    =      0.000      0.000      0.000
    SPCRFL    =      0.000      0.000      0.000
    GLOSS     =      0.000
  
```

MATERIAL

```

    TYPE      = DULLMIRROR
    TEXTURE   = SMOOTH
    AMBRFL    =      0.100      0.100      0.100
    DIFRFL    =      0.100      0.100      0.100
    SPCRFL    =      0.100      0.100      0.100
    GLOSS     =     10.000
  
```

GROUND

```

    MATL      = GRASS
  
```

```

{ moon }
  
```

LAMP

```

    LOC       =    100.000   -999.000   9000.000
    RADIUS    =    300.000
    INTENS    =      0.100      0.100      0.100
  
```

```

{ birdbath }
  
```


DEFINING A SCENE: THE .RT FILE

CYLINDER

LOC	=	0.000	0.000	0.000
DIRECT	=	0.000	0.000	1.000
RADIUS	=	80.000		
HEIGHT	=	25.000		
MATL	=	CEMENT		

RING

LOC	=	0.000	0.000	25.000
V1	=	1.000	0.000	0.000
V2	=	0.000	1.000	0.000
RAD1	=	70.000		
RAD2	=	80.000		
MATL	=	CEMENT		

CYLINDER

LOC	=	0.000	0.000	25.000
DIRECT	=	0.000	0.000	1.000
RADIUS	=	70.000		
HEIGHT	=	20.000		
MATL	=	CEMENT		

RING

LOC	=	0.000	0.000	45.000
V1	=	1.000	0.000	0.000
V2	=	0.000	1.000	0.000
RAD1	=	65.000		
RAD2	=	70.000		
MATL	=	CEMENT		

CIRCLE

LOC	=	0.000	0.000	44.500
V1	=	1.000	0.000	0.000
V2	=	0.000	1.000	0.000
RADIUS	=	68.000		
MATL	=	WATER		

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

CYLINDER

LOC	=	0.000	0.000	45.000
DIRECT	=	0.000	0.000	1.000
RADIUS	=	6.000		
HEIGHT	=	28.000		
MATL	=	CEMENT		

CONE

LOC	=	0.000	0.000	60.000
DIRECT	=	0.000	0.000	-1.000
RADIUS	=	40.000		
HEIGHT	=	15.000		
MATL	=	CEMENT		

CIRCLE

LOC	=	0.000	0.000	74.000
V1	=	1.000	0.000	0.000
V2	=	0.000	1.000	0.000
RADIUS	=	39.000		
MATL	=	WATER		

{ lake water }

PARALLELOGRAM

LOC	=	-999.000	750.000	1.000
V1	=	9999.000	0.000	0.000
V2	=	0.000	5000.000	0.000
MATL	=	LAKEWATER		

{ sidewalk }

PARALLELOGRAM

LOC	=	-299.000	160.000	1.500
V1	=	1999.000	0.000	0.000
V2	=	0.000	80.000	0.000
MATL	=	LIGHTCEMENT		

DEFINING A SCENE: THE .RT FILE

```
{ streetlamps }
```

```
CYLINDER
```

```
LOC      =  -200.000    250.000    0.000
DIRECT   =    0.000    0.000    1.000
RADIUS   =    6.000
HEIGHT   =   100.000
MATL     = BLACKANODIZED
```

```
LAMP
```

```
LOC      =  -200.000    250.000    115.000
RADIUS   =   15.000
INTENS    =    0.300    0.300    0.200
```

```
CYLINDER
```

```
LOC      =   100.000    250.000    0.000
DIRECT   =    0.000    0.000    1.000
RADIUS   =    6.000
HEIGHT   =   100.000
MATL     = BLACKANODIZED
```

```
LAMP
```

```
LOC      =   100.000    250.000    115.000
RADIUS   =   15.000
INTENS    =    0.300    0.300    0.200
```

```
CYLINDER
```

```
LOC      =   400.000    250.000    0.000
DIRECT   =    0.000    0.000    1.000
RADIUS   =    6.000
HEIGHT   =   100.000
MATL     = BLACKANODIZED
```

```
LAMP
```

```
LOC      =   400.000    250.000    115.000
RADIUS   =   15.000
INTENS    =    0.300    0.300    0.200
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

CYLINDER

LOC	=	700.000	250.000	0.000
DIRECT	=	0.000	0.000	1.000
RADIUS	=	6.000		
HEIGHT	=	100.000		
MATL	=	BLACKANODIZED		

LAMP

LOC	=	700.000	250.000	115.000
RADIUS	=	15.000		
INTENS	=	0.300	0.300	0.200

{ park bench }

CYLINDER

LOC	=	400.000	120.000	0.000
DIRECT	=	0.000	0.000	1.000
RADIUS	=	10.000		
HEIGHT	=	35.000		
MATL	=	CEMENT		

CYLINDER

LOC	=	500.000	120.000	0.000
DIRECT	=	0.000	0.000	1.000
RADIUS	=	10.000		
HEIGHT	=	35.000		
MATL	=	CEMENT		

BOX

LOC	=	380.000	90.000	35.000
V1	=	140.000	0.000	0.000
V2	=	0.000	60.000	0.000
V3	=	0.000	0.000	5.000
MATL	=	CEMENT		

{ garbage can }

CYLINDER

LOC	=	200.000	120.000	0.000
-----	---	---------	---------	-------

DEFINING A SCENE: THE .RT FILE

```
DIRECT = 0.000 0.000 1.000
RADIUS = 16.000
HEIGHT = 35.000
MATL= CEMENT
```

```
{ city }
```

```
PYRAMID
LOC = -200.000 9999.000 0.000
V1 = 400.000 0.000 0.000
V2 = 0.000 400.000 0.000
HEIGHT = 1200.000
MATL = DULLCEMENT
```

```
BOX
LOC = 790.000 9999.000 0.000
V1 = 450.000 0.000 0.000
V2 = 0.000 450.000 0.000
V3 = 0.000 0.000 2200.000
MATL = DULLCEMENT
```

```
CYLINDER
LOC = 4500.000 9999.000 0.000
DIRECT = 0.000 0.000 1.000
RADIUS = 510.000
HEIGHT = 1500.000
MATL = DULLMIRROR
```

Figure 18-1. Listing of .RT Data Files for Typical Scenes

The Ray Tracing Program

The ray-tracing program (*Raytrace.Pas*) reads in a scene-description (*.RT*) file and operates upon it to produce a *.CPR* file containing red, green, and blue color information for every pixel in the picture. This program is listed in Figure 19-1, and will be described in detail in this chapter. At the same time the scene is being ray-traced, a monochrome display (in shades of gray) is being created on the display screen. This gives you an idea of how the full-color display will look, so if there are problems with your scene description, you can make modifications without having to run *Display.Pas*. The detailed color information in the *.CPR* file is operated upon by the *Display.Pas* program, to produce a full-color picture on the screen. You need to run this program to see the capabilities of ray-tracing to produce a pleasing color picture. The *Display.Pas* file will be described in Chapter 20.

Ray-tracing Overview

The easiest way to achieve an overall understanding of how the ray-tracing program works is to start at the end of the listing with the main program. First, however, let's take a quick look at some of the constants and variables defined at the beginning of the program. The first of these is *ReflectiveLamps*. This is a boolean constant which indicates if the lamps in the program will reflect light from their surface. Obviously, if your light source is supposed to be the sun, it is far too bright to reflect any other light sources, so you should set this constant to *False*. However, for less intense light sources, it is quite possible that they will reflect part of the light cast upon them, in which case you would set the constant to *True*. The next constant is *DistEffect*, which is the fractional effect of distance upon lamp intensity. By default, it is set to 0.3. You can change it if you want the light-intensity falloff with distance to be greater or less in a particular scene. The next variable to be defined is

OutputFile, which is the name of the disk file to which the pixel color data will be sent for storage. This is followed by two real numbers, *XPitch* and *YPitch*. These parameters determine the coverage of the scene area. They are essentially the inverse of the focal length, in that the larger the focal length, the more restricted the area covered by the "camera lens," whereas the larger the values of the pitch parameters, the larger the area covered. However, unlike the focal length of a lens, these parameters can be separately set for the *x* and *y* directions, thereby permitting selective distortion of the scene. It is necessary when using the 320 x 200 graphics mode 19 to correct for improper aspect ratio because of the greater horizontal pixel distance relative to the vertical pixel distance. The 1024 x 768 mode does not require this correction because the pixels are square and the pixel distances are the same in the horizontal and vertical directions. The next parameters determine the weighing of color components for a particular scene. These are *LoclWgt*, the weighing of object color component; *ReflWgt*, the weighing of the reflected color component; *MinWgt*, the minimum weighing of colors in the scene; and *MaxWgt*, the maximum weighing of colors in the scene. These are all vectors, having values for the red, green, and blue components. There is also the integer variable *MaxDepth*, which is the maximum depth of the scene.

Next, the focal length and the viewer position information are defined. The geometry for ray-tracing is the same as the solid-modeling geometry shown in Figure 8-1. The variable *ObsPos* is the observer position (a three-dimensional vector). The variables *ObsRotate* (the observer rotate angle) and *ObsTilt* (the observer tilt angle) are *View Theta* and *View Phi* respectively. From these, it is possible to compute the variable *ViewDir* (the viewer direction from the origin) and *U* and *V* (the transformations from the scene-coordinate system to a viewer-centered coordinate system).

The sky colorings are defined. These are *HorCol* (the color of the sky at the horizon) and *ZenCol* (the color of the sky at the zenith). Then the program sets up variables for tiling with a checkerboard pattern. These are *Tile1* and *Tile2* (the two checkerboard colors) and *Tile*, the size of the side of the checkerboard square. The program sets up a maximum value of 20 for the types of material that may be defined and then defines the variables that make up a material description record. These include the name of the material type, the name of its texture type, the ambient,

diffuse, and reflected color values, and the glossiness power. An array of these material records is then created. The program then sets up a parameter for the normal ambient light intensity. Next, the maximum number of textures is set to be 6. They are *Smooth*, *Checker*, *Grit*, *Marble*, *Wood*, and *Sheetrock*. The program then establishes the number of primitive shapes as 9: *Ground*, *Lamp*, *Triangle*, *Parallelogram*, *Circles*, *Ring*, *Sphere*, *Quadratic*, *Cone*, and *Cylinder*. The maximum value for each shape is specified. There is only one ground. There can be 25 lamps, 100 triangles, 100 parallelograms, 50 circles, 50 rings, 100 spheres, 50 quadratics, 50 cones, and 50 cylinders. These numbers may be increased but are limited to available memory.

The parameters associated with each primitive object are defined. These include its position, vectors that define its extent and orientation, radii (if appropriate), intensity (for lamps), surface normal, material, texture, the dot product of the normal and the location vector, and whatever other parameters are needed to define the object and compute intersections of light rays with it. The program then sets up an array containing all of the primitive objects that may be defined. Finally, the program defines the *.RT* disk file which contains the scene definition data.

Now you're ready to deal with the main program, which by itself is quite simple. It first calls the procedure *InitNoise*. This procedure sets up the noise function to create marble and wood textures. It then calls *ClearMemory*, which initializes all of the program parameters. It then calls *ClrScr*, which clears the display screen. The program calls *GetFileName*. This procedure first displays, "Enter File Name =>". It reads whatever file name the user types in from the keyboard. If a return is entered, the program will use the file *Example1.RT* by default. (The file name is typed in without the extension; this procedure then adds the extension.) The resulting file name is stored and used throughout the program. The program calls *GetData*, passing the designated file name as a parameter. This procedure loads all of the data from the selected scene data file into the program. The program then calls the proper *InitGraphics* procedure to prepare for graphics display with whatever resolution selected for this run of the program. It then calls *InitPalette1* and *SetPalette* to set up the VGA color registers for the gray-scale display. Next, the program calls the procedure *Scan* which performs all of the ray-tracing operations and saves the resulting scene data to the output file. The program calls the Turbo Pascal *Dispose*

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

procedure to free up memory allocated for the noise array and then sets the noise array address to nil. Then it calls the procedure *ExitGraphics* to exit from the graphics mode, after which the program terminates.

```
{
```

Recursive Ray-tracing Program
Renders Reflective Objects
Christopher D. Watkins

```
}
```

Uses

Dos, Crt;

{\$I Math2.Inc}

{\$I Graph2.Inc}

```
{
```

Declare Constants and Variables

```
}
```

Const

ReflectiveLamps = True; { will the surface of a lamp reflect light? }

DistEffect = 0.3; { percentage of distance effect on lamp }

Type

Name = String[32];

Var

OutputFile : Name; { Stats }

XPitch, YPitch : Real;

THE RAY TRACING PROGRAM

```
LoclWgt      : TDA;  { Environment }
ReflWgt      : TDA;
MinWgt       : TDA;
MaxWgt       : TDA;
MaxDepth     : Integer;
```

```
FocalLength  : Real; { Observer }
ObsPos       : TDA;
ObsRotate    : Real;
ObsTilt      : Real;
ViewDir      : TDA;
U, V        : TDA;
```

```
HorCol       : TDA;  { Sky horizon to zenith coloration }
ZenCol       : TDA;
```

```
Tile1        : TDA;  { Checkerboard tiling coloration }
Tile2        : TDA;
Tile         : Real;
```

```
Const
  MaxMaterial = 20;      { Maximum Types of Materials }
```

```
Type
  MaterialList = Record
    MType : Name;
    Textur : Name;
    AmbRfl : TDA;
    DifRfl : TDA;
    SpcRfl : TDA;
    Gloss : Real
  End;
```

```
Var
  Mat1 : Array[1..MaxMaterial] Of MaterialList;
```

```
Var
  AmbIntens : TDA;
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

Const

MaxTexture = 6; { Maximum Number of Textures }

Smooth = 1; { Texture Number }

Checker = 2;

Grit = 3;

Marble = 4;

Wood = 5;

Sheetrock = 6;

MaxShapeType = 9; { Maximum Number of Shape Types }

Ground = 0; { Ground is Shape Type 0 - a high speed calc }

Lamp = 1; { Shape Type Number }

Triangle = 2;

Parallelogram = 3;

Circles = 4;

Ring = 5;

Sphere = 6;

Quadratic = 7;

Cone = 8;

Cylinder = 9;

MaxLamp = 25;

MaxTriangle = 100; { Max Count of Objects for any one Shape Type }

MaxParallelogram = 100;

MaxCircles = 50;

MaxRing = 50;

MaxSphere = 100;

MaxQuadratic = 50;

MaxCone = 50;

MaxCylinder = 50;

Type

GroundList = Record

MtlNum : Integer;

TexNum : Integer

THE RAY TRACING PROGRAM

End;

LampList = Record

Loc : TDA;
Rad : Real;
RadSqr : Real;
Intens : TDA

End;

TriangleList = Record

Loc : TDA;
v1 : TDA;
v2 : TDA;
Norm : TDA;
NdotLoc : Real;
MtlNum : Integer;
TexNum : Integer

End;

ParallelogramList = Record

Loc : TDA;
v1 : TDA;
v2 : TDA;
Norm : TDA;
NdotLoc : Real;
MtlNum : Integer;
TexNum : Integer

End;

CirclesList = Record

Loc : TDA;
v1 : TDA;
v2 : TDA;
Norm : TDA;
NdotLoc : Real;
Radius : Real;
MtlNum : Integer;
TexNum : Integer

End;

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

RingList = Record

Loc : TDA;
v1 : TDA;
v2 : TDA;
Norm : TDA;
NdotLoc : Real;
Rad1 : Real;
Rad2 : Real;
MtlNum : Integer;
TexNum : Integer

End;

SphereList = Record { special case of Quadratic }

Loc : TDA;
Rad : Real;
RadSqr : Real;
MtlNum : Integer;
TexNum : Integer

End;

QuadraticList = Record

Loc : TDA;
Direct : TDA;
cos1 : Real;
sin1 : Real;
cos2 : Real;
sin2 : Real;
c1 : Real;
c2 : Real;
c3 : Real;
c4 : Real;
c5 : Real;
xmin : Real;
xmax : Real;
ymin : Real;
ymax : Real;
zmin : Real;
zmax : Real;
MtlNum : Integer;
TexNum : Integer

End;

THE RAY TRACING PROGRAM

ConeList = QuadraticList;

CylinderList = QuadraticList;

Var

ObjCnt : Array[0..MaxShapeType] Of Integer;

Gnd : GroundList;

Lmp : Array[1..MaxLamp] Of LampList;

Tri : Array[1..MaxTriangle] Of TriangleList;

Para : Array[1..MaxParallelogram] Of ParallelogramList;

Cir : Array[1..MaxCircles] Of CirclesList;

Rng : Array[1..MaxRing] Of RingList;

Sphr : Array[1..MaxSphere] Of SphereList;

Quad : Array[1..MaxQuadratic] Of QuadraticList;

Con : Array[1..MaxCone] Of ConeList;

Cyl : Array[1..MaxCylinder] Of CylinderList;

Var

DiskFile : File Of Char;

TextDiskFile : Text;

{

Clear all Variables

ClearMemory - clear all variables

}

Procedure ClearMemory;

Procedure ClearQuadratic (Var List : QuadraticList);

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
Begin
  With List Do
    Begin
      VecNull(Loc);
      VecNull(Direct);
      cos1 := 0.0;
      sin1 := 0.0;
      cos2 := 0.0;
      sin2 := 0.0;
      c1 := 0.0;
      c2 := 0.0;
      c3 := 0.0;
      c4 := 0.0;
      c5 := 0.0;
      xmin := 0.0;
      xmax := 0.0;
      ymin := 0.0;
      ymax := 0.0;
      zmin := 0.0;
      zmax := 0.0;
      MtlNum := 0;
      TexNum := 0
    End;
  End;
```

```
Var
  i : Integer;
```

```
Begin
  OutputFile := '';
  XRes := 0;
  YRes := 0;
  XPitch := 0.0;
  YPitch := 0.0;

  VecNull(LocWgt);
  VecNull(RefWgt);
  VecNull(MinWgt);
  VecNull(MaxWgt);
  MaxDepth := 0;
```


THE RAY TRACING PROGRAM

```
FocalLength := 0.0;
VecNull(ObsPos);
ObsRotate := 0.0;
ObsTilt := 0.0;
VecNull(ViewDir);
VecNull(U);
VecNull(V);

VecNull(HorCol);
VecNull(ZenCol);

VecNull(Tile1);
VecNull(Tile2);
Tile := 0.0;

For i := 1 To MaxMaterial Do
  With Matl[i] Do
    Begin
      MType := '';
      Textur := '';
      VecNull(AmbRfl);
      VecNull(DifRfl);
      VecNull(SpcRfl);
      Gloss := 0.0
    End;

VecNull(AmbIntens);

ObjCnt[Ground] := 1; { Count for a particular object }
For i := 1 To MaxShapeType Do
  ObjCnt[i] := 0; { Note : count for Ground is always 1 }

With Gnd Do
  Begin
    MtlNum := 0;
    TexNum := 0
  End;

For i := 1 To MaxLamp Do
  With Lmp[i] Do
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
Begin
  VecNull(Loc);
  Rad      := 0.0;
  RadSqr   := 0.0;
  VecNull(Intens)
End;

For i := 1 To MaxTriangle Do
  With Tri[i] Do
    Begin
      VecNull(Loc);
      VecNull(v1);
      VecNull(v2);
      VecNull(Norm);
      NdotLoc := 0.0;
      MtlNum  := 0;
      TexNum  := 0
    End;
  End;

For i := 1 To MaxParallelogram Do
  With Para[i] Do
    Begin
      VecNull(Loc);
      VecNull(v1);
      VecNull(v2);
      VecNull(Norm);
      NdotLoc := 0.0;
      MtlNum  := 0;
      TexNum  := 0
    End;
  End;

For i := 1 To MaxCircles Do
  With Cir[i] Do
    Begin
      VecNull(Loc);
      VecNull(v1);
      VecNull(v2);
      VecNull(Norm);
      NdotLoc := 0.0;
      Radius  := 0.0;
    End;
  End;
End;
```


THE RAY TRACING PROGRAM

```
        MtlNum := 0;
        TexNum := 0
    End;

    For i := 1 To MaxRing Do
        With Rng[i] Do
            Begin
                VecNull(Loc);
                VecNull(v1);
                VecNull(v2);
                VecNull(Norm);
                NdotLoc := 0.0;
                Rad1 := 0.0;
                Rad2 := 0.0;
                MtlNum := 0;
                TexNum := 0
            End;
        End;

    For i := 1 To MaxSphere Do
        With Spshr[i] Do
            Begin
                VecNull(Loc);
                Rad := 0.0;
                RadSqr := 0.0;
                MtlNum := 0;
                TexNum := 0
            End;
        End;

    For i := 1 To MaxQuadratic Do
        ClearQuadratic(Quad[i]);

    For i := 1 To MaxCone Do
        ClearQuadratic(Con[i]);

    For i := 1 To MaxCylinder Do
        ClearQuadratic(Cyl[i])

End;
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
{  


|                  |
|------------------|
| Load a *.RT File |
|------------------|

  
}
```

```
Var  
  MtlCount : Integer; { Number of Materials Loaded }
```

```
Procedure GetData(FileName : Name);
```

```
Procedure GetViewDir(Angl, Tilt : Real; Var View, U, V : TDA);
```

```
Var  
  Phi, Theta : Real;  
  x, y, z     : Real;
```

```
Begin
```

```
  Phi := Radians( Angl );  
  Theta := Radians( Tilt );
```

```
  x := Cos( Theta ) * Sin( Phi );  
  y := Cos( Theta ) * Cos( Phi );  
  z := -Sin( Theta );  
  Vec(x, y, z, View);
```

```
  x := Cos( Phi );  
  y := -Sin( Phi );  
  z := 0;  
  Vec(x, y, z, u);
```

```
  x := Sin( Theta ) * Sin( Phi );  
  y := Sin( Theta ) * Cos( Phi );  
  z := Cos( Theta );  
  Vec(x, y, z, v)
```

```
End;
```

```
Var
```


THE RAY TRACING PROGRAM

```
Buf, Buf2 : String;
dummy      : Integer;
MtlName     : Name;

Procedure LoadTDA(Var A : TDA);

  Var
    i : Integer;

  Begin
    ReadLn(TextDiskFile, Buf2);
    For i := 0 To 2 Do
      Val(Copy(Buf2, 14 + i * 9, 8), A[i], dummy)
    End;

Procedure LoadReal(Var a : Real);

  Begin
    ReadLn(TextDiskFile, Buf2);
    Val(Copy(Buf2, 14, 8), a, dummy)
  End;

Procedure LoadInteger(Var a : Integer);

  Begin
    ReadLn(TextDiskFile, Buf2);
    Val(Copy(Buf2, 14, 8), a, dummy)
  End;

Procedure LoadText(Var a : Name);

  Begin
    ReadLn(TextDiskFile, Buf2);
    a := Copy(Buf2, 14, 30)
  End;

Procedure GetMatlNum (Mat : String; Var MatNum : Integer);

  Var
    { Associate Materials to Objects }
    i : Integer;
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
Begin
  For i := 1 To MtlCount Do
    If (Matl[i].MType = Mat) Then MatNum := i
  End;
```

```
Procedure GetTexNum(Tex : String; Var TexNum : Integer);
```

```
Begin
  If Tex = 'SMOOTH' Then
    TexNum := Smooth
  Else
    If Tex = 'CHECKER' Then
      TexNum := Checker
    Else
      If Tex = 'GRIT' Then
        TexNum := Grit
      Else
        If Tex = 'MARBLE' Then
          TexNum := Marble
        Else
          If Tex = 'WOOD' Then
            TexNum := Wood
          Else
            If Tex = 'SHEETROCK' Then
              TexNum := Sheetrock
            End;
          End;
        End;
      End;
    End;
  End;
```

```
Procedure OrientQuadratic(Var List : QuadraticList);
```

```
Var
  Temp      : Real;
  NewDirect : TDA;
```

```
Begin
  With List Do
    Begin
      VecNormalize(Direct);
      Temp := Sqr(Direct[0]) + Sqr(Direct[2]);
```


THE RAY TRACING PROGRAM

```
      If (Temp = 0) Then
        cos1 := 1
      Else
        cos1 := Direct[2] / Sqrt(Temp);
        sin1 := Sqrt(1 - Sqr(cos1));
        NewDirect[0] := Direct[0] * cos1 - Direct[2] * sin1;
        NewDirect[1] := Direct[1];
        NewDirect[2] := Direct[0] * sin1 + Direct[2] * cos1;
        Temp := Sqr(NewDirect[1]) + Sqr(NewDirect[2]);
        If (Temp = 0) Then
          cos2 := 1
        Else
          cos2 := NewDirect[1] / Sqrt(Temp);
          sin2 := Sqrt(1 - Sqr(cos2))
        End
      End;

End;

Var
  Rad, Hgt : Real;

  ShapeLoc, TempLoc, vec1, vec2, vec3 : TDA;
  MtlNumber, TexNumber : Integer;

Begin
  MtlCount := 0;

  Assign(TextDiskFile, FileName);
  Reset(TextDiskFile);

  Repeat

    ReadLn(TextDiskFile, Buf);

    If Buf = 'STATS' Then
      Begin
        LoadText(OutputFile);
        LoadInteger(XRes);
        LoadInteger(YRes);
        LoadReal(XPitch);
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
        LoadReal(YPitch)
    End;

    If Buf = 'ENVIRONMENT' Then
    Begin
        LoadTDA(LocIWgt);
        LoadTDA(RefIWgt);
        LoadTDA(MinWgt);
        LoadTDA(MaxWgt);
        LoadInteger(MaxDepth)
    End;

    If Buf = 'OBSERVER' Then
    Begin
        LoadReal(FocalLength);
        LoadTDA(ObsPos);
        LoadReal(ObsRotate);
        LoadReal(ObsTilt);
        GetViewDir(ObsRotate, ObsTilt, ViewDir, U, V)
    End;

    If Buf = 'SKY' Then
    Begin
        LoadTDA(HorCol);
        LoadTDA(ZenCol)
    End;

    If Buf = 'CHECK' Then
    Begin
        LoadTDA(Tile1);
        LoadTDA(Tile2);
        LoadReal(Tile)
    End;

    If Buf = 'MATERIAL' Then
    Begin
        MtlCount := MtlCount + 1;
        With Matl[MtlCount] Do
            Begin
                LoadText(MType);
```


THE RAY TRACING PROGRAM

```
        LoadText(Textur);
        LoadTDA(AmbRfl);
        LoadTDA(DifRfl);
        LoadTDA(SpcRfl);
        LoadReal(Gloss)
    End
End;

If Buf = 'AMBIENTLIGHT' Then
    Begin
        LoadTDA(AmbIntens)
    End;

    If Buf = 'GROUND' Then
        Begin
            With Gnd Do
                Begin
                    LoadText(MtlName);
                    GetMatlNum(MtlName, MtlNum);
                    GetTexNum(Matl[MtlNum].Textur, TexNum)
                End
            End;

            If Buf = 'LAMP' Then
                Begin
                    ObjCnt[Lamp] := ObjCnt[Lamp] + 1;
                    With Lmp[ ObjCnt[Lamp] ] Do
                        Begin
                            LoadTDA(Loc);
                            LoadReal(Rad);
                            RadSqr := Sqr(Rad);
                            LoadTDA(Intens)
                        End
                    End;

                    If Buf = 'TRIANGLE' Then
                        Begin
                            ObjCnt[Triangle] := ObjCnt[Triangle] + 1;
                            With Tri[ ObjCnt[Triangle] ] Do
                                Begin
```



```

        LoadTDA(Loc);
        LoadTDA(v1);
        LoadTDA(v2);
        VecCross(v1, v2, Norm);
        VecNormalize(Norm);
        NdotLoc := VecDot(Norm, Loc);
        LoadText(MtlName);
        GetMatlNum(MtlName, MtlNum);
        GetTexNum(Matl[MtlNum].Textur, TexNum)
    End
End;

If Buf = 'PARALLELOGRAM' Then
Begin
    ObjCnt[Parallelogram] := ObjCnt[Parallelogram] + 1;
    With Para[ ObjCnt[Parallelogram] ] Do
        Begin
            LoadTDA(Loc);
            LoadTDA(v1);
            LoadTDA(v2);
            VecCross(v1, v2, Norm);
            VecNormalize(Norm);
            NdotLoc := VecDot(Norm, Loc);
            LoadText(MtlName);
            GetMatlNum(MtlName, MtlNum);
            GetTexNum(Matl[MtlNum].Textur, TexNum)
        End
    End;

If Buf = 'CIRCLE' Then
Begin
    ObjCnt[Circles] := ObjCnt[Circles] + 1;
    With Cir[ ObjCnt[Circles] ] Do
        Begin
            LoadTDA(Loc);
            LoadTDA(v1);
            VecNormalize(v1);
            LoadTDA(v2);
            VecNormalize(v2);
            VecCross(v1, v2, Norm);

```


THE RAY TRACING PROGRAM

```
VecNormalize(Norm);
NdotLoc := VecDot(Norm, Loc);
LoadReal(Radius);
LoadText(MtlName);
GetMatlNum(MtlName, MtlNum);
GetTexNum(Matl[MtlNum].Textur, TexNum)
End
End;

If Buf = 'RING' Then
Begin
ObjCnt[Ring] := ObjCnt[Ring] + 1;
With Rng[ ObjCnt[Ring] ] Do
Begin
LoadTDA(Loc);
LoadTDA(v1);
VecNormalize(v1);
LoadTDA(v2);
VecNormalize(v2);
VecCross(v1, v2, Norm);
VecNormalize(Norm);
NdotLoc := VecDot(Norm, Loc);
LoadReal(Rad1);
LoadReal(Rad2);
LoadText(MtlName);
GetMatlNum(MtlName, MtlNum);
GetTexNum(Matl[MtlNum].Textur, TexNum)
End
End;

If Buf = 'SPHERE' Then
Begin
ObjCnt[Sphere] := ObjCnt[Sphere] + 1;
With Sphe[ ObjCnt[Sphere] ] Do
Begin
LoadTDA(Loc);
LoadReal(Rad);
RadSqr := Sqr(Rad);
LoadText(MtlName);
GetMatlNum(MtlName, MtlNum);
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
        GetTexNum(Mat1[Mt1Num].Textur, TexNum)
    End
End;

If Buf = 'QUADRATIC' Then {  $C1 x^2 + C2 y^2 + C3 z^2 + C5 y = C4$  }
Begin
    ObjCnt[Quadratic] := ObjCnt[Quadratic] + 1;
    With Quad[ ObjCnt[Quadratic] ] Do
        Begin
            LoadTDA(Loc);
            LoadTDA(Direct);
            OrientQuadratic(Quad[ ObjCnt[Quadratic] ]);
            LoadReal(c1);
            LoadReal(c2);
            LoadReal(c3);
            LoadReal(c4);
            LoadReal(c5);
            LoadReal(xmin);
            LoadReal(xmax);
            LoadReal(ymin);
            LoadReal(ymax);
            LoadReal(zmin);
            LoadReal(zmax);
            LoadText(Mt1Name);
            GetMat1Num(Mt1Name, Mt1Num);
            GetTexNum(Mat1[Mt1Num].Textur, TexNum)
        End
    End;
End;

If Buf = 'CONE' Then {  $h^2 x^2 - 2r^2 y^2 + h^2 z^2 = 0$  }
Begin
    ObjCnt[Cone] := ObjCnt[Cone] + 1;
    With Con[ ObjCnt[Cone] ] Do
        Begin
            LoadTDA(Loc);
            LoadTDA(Direct);
            OrientQuadratic(Con[ ObjCnt[Cone] ]);
            LoadReal(Rad);
            LoadReal(Hgt);
            xmin := - Rad;
```


THE RAY TRACING PROGRAM

```
    zmin := - Rad;
    ymin := - Hgt;
    xmax := Rad;
    zmax := Rad;
    ymax := 0.0;
    c1 := Sqr(Rad);
    c2 := -Sqr(c1);
    c3 := Sqr(Hgt);
    c1 := c1 * c3;
    c3 := c1;
    c4 := 0.0;
    c5 := 0.0;
    LoadText(Mt1Name);
    GetMat1Num(Mt1Name, Mt1Num);
    GetTexNum(Mat1[Mt1Num].Textur, TexNum)
End
End;

If Buf = 'CYLINDER' Then
Begin
    ObjCnt[Cylinder] := ObjCnt[Cylinder] + 1;
    With Cyl[ ObjCnt[Cylinder] ] Do
    Begin
        LoadTDA(Loc);
        LoadTDA(Direct);
        OrientQuadratic(Cyl[ ObjCnt[Cylinder] ]);
        LoadReal(Rad);
        LoadReal(Hgt);
        xmin := - Rad;
        zmin := - Rad;
        ymin := 0;
        xmax := Rad;
        zmax := Rad;
        ymax := Hgt;
        c1 := 1.0;
        c2 := 0.0;
        c3 := 1.0;
        c4 := Sqr(Rad);
        c5 := 0.0;
        LoadText(Mt1Name);
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
        GetMatlNum(MtlName, MtlNum);
        GetTexNum(Matl[MtlNum].Textur, TexNum)
    End
End;

If Buf = 'BOX' Then
Begin
    LoadTDA(ShapeLoc);
    LoadTDA(vec1);
    LoadTDA(vec2);
    LoadTDA(vec3);
    LoadText(MtlName);
    GetMatlNum(MtlName, MtlNumber);
    GetTexNum(Matl[MtlNumber].Textur, TexNumber);

    ObjCnt[Parallelogram] := ObjCnt[Parallelogram] + 1;
    With Para[ ObjCnt[Parallelogram] ] Do
        Begin
            VecCopy(ShapeLoc, Loc);
            VecCopy(vec1, v1);
            VecCopy(vec3, v2);
            VecCross(v1, v2, Norm);
            VecNormalize(Norm);
            NdotLoc := VecDot(Norm, Loc);
            MtlNum := MtlNumber;
            TexNum := TexNumber
        End;

    ObjCnt[Parallelogram] := ObjCnt[Parallelogram] + 1;
    With Para[ ObjCnt[Parallelogram] ] Do
        Begin
            VecCopy(ShapeLoc, Loc);
            VecCopy(vec3, v1);
            VecCopy(vec1, v2);
            VecCross(v1, v2, Norm);
            VecNormalize(Norm);
            Vec(0, vec2[1], 0, TempLoc);
            VecAdd(TempLoc, Loc, Loc);
            NdotLoc := VecDot(Norm, Loc);
            MtlNum := MtlNumber;
```


THE RAY TRACING PROGRAM

```
    TexNum := TexNumber  
End;
```

```
ObjCnt[Parallelogram] := ObjCnt[Parallelogram] + 1;  
With Para[ ObjCnt[Parallelogram] ] Do  
Begin  
    VecCopy(ShapeLoc, Loc);  
    VecCopy(vec3, v1);  
    VecCopy(vec2, v2);  
    VecCross(v1, v2, Norm);  
    VecNormalize(Norm);  
    NdotLoc := VecDot(Norm, Loc);  
    MtlNum := MtlNumber;  
    TexNum := TexNumber  
End;
```

```
ObjCnt[Parallelogram] := ObjCnt[Parallelogram] + 1;  
With Para[ ObjCnt[Parallelogram] ] Do  
Begin  
    VecCopy(ShapeLoc, Loc);  
    VecCopy(vec2, v1);  
    VecCopy(vec3, v2);  
    VecCross(v1, v2, Norm);  
    VecNormalize(Norm);  
    Vec(vec1[0], 0, 0, TempLoc);  
    VecAdd(TempLoc, Loc, Loc);  
    NdotLoc := VecDot(Norm, Loc);  
    MtlNum := MtlNumber;  
    TexNum := TexNumber  
End;
```

```
ObjCnt[Parallelogram] := ObjCnt[Parallelogram] + 1;  
With Para[ ObjCnt[Parallelogram] ] Do  
Begin  
    VecCopy(ShapeLoc, Loc);  
    VecCopy(vec2, v1);  
    VecCopy(vec1, v2);  
    VecCross(v1, v2, Norm);  
    VecNormalize(Norm);  
    NdotLoc := VecDot(Norm, Loc);
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
        MtlNum := MtlNumber;
        TexNum := TexNumber
    End;

    ObjCnt[Parallelogram] := ObjCnt[Parallelogram] + 1;
    With Para[ ObjCnt[Parallelogram] ] Do
        Begin
            VecCopy(ShapeLoc, Loc);
            VecCopy(vec1, v1);
            VecCopy(vec2, v2);
            VecCross(v1, v2, Norm);
            VecNormalize(Norm);
            Vec(0, 0, vec3[2], TempLoc);
            VecAdd(TempLoc, Loc, Loc);
            NdotLoc := VecDot(Norm, Loc);
            MtlNum := MtlNumber;
            TexNum := TexNumber
        End
    End;

    If Buf = 'PYRAMID' Then
        Begin
            LoadTDA(ShapeLoc);
            LoadTDA(vec1);
            LoadTDA(vec2);
            LoadReal(Hgt);
            LoadText(MtlName);
            GetMatlNum(MtlName, MtlNumber);
            GetTexNum(Matl[MtlNumber].Textur, TexNumber);

            ObjCnt[Parallelogram] := ObjCnt[Parallelogram] + 1;
            With Para[ ObjCnt[Parallelogram] ] Do
                Begin
                    VecCopy(ShapeLoc, Loc);
                    VecCopy(vec2, v1);
                    VecCopy(vec1, v2);
                    VecCross(v1, v2, Norm);
                    VecNormalize(Norm);
                    NdotLoc := VecDot(Norm, Loc);
                    MtlNum := MtlNumber;
```


THE RAY TRACING PROGRAM

```
    TexNum := TexNumber
End;

ObjCnt[Triangle] := ObjCnt[Triangle] + 1;
With Tri[ ObjCnt[Triangle] ] Do
Begin
    VecCopy(ShapeLoc, Loc);
    VecCopy(vec1, v1);
    Vec(0.5*vec1[0], 0.5*vec2[1], Hgt, v2);
    VecCross(v1, v2, Norm);
    VecNormalize(Norm);
    NdotLoc := VecDot(Norm, Loc);
    MtlNum := MtlNumber;
    TexNum := TexNumber
End;

ObjCnt[Triangle] := ObjCnt[Triangle] + 1;
With Tri[ ObjCnt[Triangle] ] Do
Begin
    Loc[0] := ShapeLoc[0] + vec1[0];
    Loc[1] := ShapeLoc[1] + vec2[1];
    Loc[2] := ShapeLoc[2];
    VecScalMult(-1.0, vec1, v1);
    Vec(-0.5*vec1[0], -0.5*vec2[1], Hgt, v2);
    VecCross(v1, v2, Norm);
    VecNormalize(Norm);
    NdotLoc := VecDot(Norm, Loc);
    MtlNum := MtlNumber;
    TexNum := TexNumber
End;

ObjCnt[Triangle] := ObjCnt[Triangle] + 1;
With Tri[ ObjCnt[Triangle] ] Do
Begin
    Loc[0] := ShapeLoc[0] + vec1[0];
    Loc[1] := ShapeLoc[1] + vec2[1];
    Loc[2] := ShapeLoc[2];
    Vec(-0.5*vec1[0], -0.5*vec2[1], Hgt, v1);
    VecScalMult(-1.0, vec2, v2);
    VecCross(v1, v2, Norm);
```



```

        VecNormalize(Norm);
        NdotLoc := VecDot(Norm, Loc);
        MtlNum := MtlNumber;
        TexNum := TexNumber
    End;

    ObjCnt[Triangle] := ObjCnt[Triangle] + 1;
    With Tri[ ObjCnt[Triangle] ] Do
    Begin
        VecCopy(ShapeLoc, Loc);
        Vec(0.5*vec1[0], 0.5*vec2[1], Hgt, v1);
        VecCopy(vec2, v2);
        VecCross(v1, v2, Norm);
        VecNormalize(Norm);
        NdotLoc := VecDot(Norm, Loc);
        MtlNum := MtlNumber;
        TexNum := TexNumber
    End
End;

Until EOF(TextDiskFile);

Close(TextDiskFile)
End;

{
    Calculate Directions for Reflected Rays
}

CalcDirOfRef1Ray - calculate the direction of a reflected ray
}

Procedure CalcDirOfSpecRef1Ray(Dir, SrfNrm : TDA; Var Ref1Ray : TDA);

Var
    Tmp : Real;
```


THE RAY TRACING PROGRAM

```
Begin
  Tmp := -2.0 * VecDot(Dir, SrfNrm);
  VecAddScalMult(Tmp, SrfNrm, Dir, ReflRay)
End;
```

```
{
```

Intersection of Ray with Objects

GetIntrPt - find the intersection point given a point, direction and dist
GetSrfNrm - find the surface normal given the point of intersection
Intersect - determine if an object has been hit by a ray
ShootRay - check ray Intersect against all objects and return closest

```
}
```

```
Procedure GetIntrPt (Pt, Dir : TDA; Dist : Real; Var IntrPt : TDA);
```

```
Begin
  VecAddScalMult(Dist, Dir, Pt, IntrPt)
End;
```

```
Procedure Rotate(vect : TDA; cos1, sin1, cos2, sin2 : Real; Var result : TDA);
```

```
Var
  temp : Real;
```

```
Begin
  result[0] := vect[0] * cos1 - vect[2] * sin1;
  temp      := vect[0] * sin1 + vect[2] * cos1;
  result[1] := - vect[1] * cos2 + temp * sin2;
  result[2] := - vect[1] * sin2 - temp * cos2
End;
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
Procedure RevRotate(vect : TDA; cos1, sin1, cos2, sin2 : Real; Var result  
: TDA);
```

```
Var  
    temp : Real;
```

```
Begin  
    result[1] := vect[1] * cos2 - vect[2] * sin2;  
    temp      := vect[1] * sin2 + vect[2] * cos2;  
    result[0] := - vect[0] * cos1 + temp * sin1;  
    result[2] := - vect[0] * sin1 - temp * cos1  
End;
```

```
Procedure GetSrfNrm(Shp, Obj : Integer; IntrPt : TDA; Var SrfNrm : TDA);
```

```
Procedure QuadraticSrfNrm(Var List : QuadraticList);
```

```
Var  
    NewPos, NewDir : TDA;
```

```
Begin  
    With List Do  
        Begin  
            VecSub(IntrPt, Loc, NewPos);  
            If ((Direct[0] = 0) And (Direct[1] = 1) And (Direct[2]  
= 0)) Then  
                Begin  
                    SrfNrm[0] := c1 * NewPos[0];  
                    SrfNrm[1] := c2 * NewPos[1];  
                    SrfNrm[2] := c3 * NewPos[2]  
                End  
            Else  
                Begin  
                    Rotate(NewPos, cos1, sin1, -cos2, sin2,  
NewPos);  
                    NewDir[0] := c1 * NewPos[0];  
                    NewDir[1] := c2 * NewPos[1];  
                    NewDir[2] := c3 * NewPos[2];  
                    RevRotate(NewDir, -cos1, sin1, -cos2, sin2,  
SrfNrm)
```


THE RAY TRACING PROGRAM

```

        End;
        VecNormalize(SrfNrm)
    End
End;

Begin
    Case Shp Of
        Ground:      Vec(0, 0, 1, SrfNrm);

        Lamp:         With Lmp[Obj] Do { lamp = |IntrPt - Loc| }
            Begin
                VecSub(IntrPt, Loc, SrfNrm);
                VecNormalize(SrfNrm)
            End;

        Triangle:     With Tri[Obj] Do { triangle = |v1 X v2| }
            VecCopy(Norm, SrfNrm);

        Parallelogram: With Para[Obj] Do { parallelogram = |v1 X v2| }
            VecCopy(Norm, SrfNrm);

        Circles:      With Cir[Obj] Do { Circles = |v1 X v2| }
            VecCopy(Norm, SrfNrm);

        Ring:         With Rng[Obj] Do { ring = |v1 X v2| }
            VecCopy(Norm, SrfNrm);

        Sphere:       With Sphr[Obj] Do { sphere = |IntrPt - Loc| }
            Begin
                VecSub(IntrPt, Loc, SrfNrm);
                VecNormalize(SrfNrm)
            End;

        Quadratic:    QuadraticSrfNrm(Quad[Obj]);

        Cone:         QuadraticSrfNrm(Con[Obj]);

        Cylinder:     QuadraticSrfNrm(Cyl[Obj])
    End
End;
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

Const

Small = 1E-03;

Function Intersect(Pt, Dir : TDA; Shp, Obj : Integer) : Real;

Var

IntrPoint : TDA;

delta : TDA;

dot : Real;

pos1, pos2 : Real;

gu, gv : FDA;

Temp : TDA;

b, c, t : Real;

t1, t2 : Real;

disc, sroot : Real;

rad : Real;

Procedure SetUpTriangle(p1, p2 : Byte);

Begin

With Tri[Obj] Do

Begin

gu[0] := - delta[p1];

gv[0] := - delta[p2];

gu[1] := v1[p1] - delta[p1];

gv[1] := v1[p2] - delta[p2];

gu[2] := v2[p1] - delta[p1];

gv[2] := v2[p2] - delta[p2]

End

End;

Procedure SetUpParallelogram(p1, p2 : Byte);

Begin

With Para[Obj] Do

Begin

gu[0] := - delta[p1];

gv[0] := - delta[p2];

THE RAY TRACING PROGRAM

```

    gu[1] := v1[p1] - delta[p1];
    gv[1] := v1[p2] - delta[p2];
    gu[2] := v2[p1] + v1[p1] - delta[p1];
    gv[2] := v2[p2] + v1[p2] - delta[p2];
    gu[3] := v2[p1] - delta[p1];
    gv[3] := v2[p2] - delta[p2]
End
End;

Function EvenCrossings(Sides : Integer) : Boolean;

Var
    i, j      : Byte;
    crossings : Word;

Begin
    crossings := 0;
    For i := 0 To Sides-1 Do
        Begin
            j := (i + 1) Mod Sides;
            If (((gv[i] < 0) And (gv[j] >= 0)) Or
                ((gv[j] < 0) And (gv[i] >= 0))) Then
                If ((gu[i] >= 0) And (gu[j] >= 0)) Then
                    crossings := crossings + 1
                Else
                    If ((gu[i] >= 0) Or (gu[j] >= 0)) Then
                        If (gu[i] - gv[i] * (gu[j] - gu[i]) / (gv[j] -
                                                                    gv[i]) > 0) Then
                            crossings := crossings + 1
                        End;
                    End;
                End;
            If ((crossings Mod 2) = 0) Then
                EvenCrossings := True
            Else
                EvenCrossings := False
            End;
        End;
    End;

Procedure QuadraticIntersectionCheck;

Begin
    If (Not(t1 > Small) And Not(t2 > Small)) Then

```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
        Intersect := -1.0
    Else
        Begin
            If (t1 > t2) Then
                Begin
                    If (t2 < Small) Then t2 := t1
                End
            Else
                Begin
                    If (t1 > Small) Then t2 := t1
                End;
            Intersect := t2
        End
    End;
```

Procedure QuadraticShapes(Var List : QuadraticList);

```
Var
    i          : Integer;
    NewLoc, NewDir : TDA;
    a, disc     : Real;
    loc2, loc1  : TDA;
```

Function PointOutOfBounds(Point : TDA) : Boolean;

```
    Begin
        With List Do
            Begin
                If ((Point[0] < xmin) Or (Point[0] > xmax) Or
                    (Point[1] < ymin) Or (Point[1] > ymax) Or
                    (Point[2] < zmin) Or (Point[2] > zmax)) Then
                    PointOutOfBounds := True
                Else
                    PointOutOfBounds := False
                End
            End
        End;
    End;
```

```
Begin
    With List Do
        Begin
```


THE RAY TRACING PROGRAM

```
VecSub(Pt, Loc, NewLoc);
If ((Direct[0] = 0) And (Direct[1] = 1) And (Direct[2]
                                         = 0)) Then
    VecCopy(Dir, NewDir)
Else
    Begin
        Rotate(Dir, cos1, sin1, cos2, sin2, NewDir);
        Rotate(NewLoc, cos1, sin1, cos2, sin2, NewLoc)
    End;
If (c5 = 0) Then
    Begin
        c := -c4 + c1 * Sqr(NewLoc[0]) +
              c2 * Sqr(NewLoc[1]) +
              c3 * Sqr(NewLoc[2]);
        b := 2 * (c1 * NewLoc[0] * NewDir[0] +
                  c2 * NewLoc[1] * NewDir[1] +
                  c3 * NewLoc[2] * NewDir[2]);
        a := c1 * Sqr(NewDir[0]) + c2 * Sqr(NewDir[1])
              + c3 * Sqr(NewDir[2])
    End
Else
    Begin
        c := -c4 + c1 * Sqr(NewLoc[0]) - c5 * NewLoc[1] +
              c3 * Sqr(NewLoc[2]);
        b := 2 * (c1 * NewLoc[0] * NewDir[0] - c5 *
                  NewDir[1] + c3 * NewLoc[2] * NewDir[2]);
        a := c1 * Sqr(NewDir[0]) + c3 * Sqr(NewDir[2])
    End;
disc := Sqr(b) - 4.0 * a * c;

If (disc < 0) Then
    Intersect := -1.0
Else
    Begin
        sroot := Sqrt(disc);
        t2 := (-b+sroot) / (a+a);
        GetIntrPt(NewLoc, NewDir, t2, loc2);
        If PointOutOfBounds(loc2) Then
            t2 := -1.0;
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
        t1 := (-b-sroot) / (a+a);
        GetIntrPt(NewLoc, NewDir, t1, loc1);
        If PointOutOfBounds(loc1) Then
            t1 := -1.0;
        QuadraticIntersectionCheck
    End
End
End;

Begin
    Case Shp Of
        Ground:   If (Dir[2] = 0.0) Then
                    Intersect := -1.0
                Else
                    Begin
                        t := - Pt[2] / Dir[2];
                        If (t > Small) Then
                            Intersect := t
                        Else
                            Intersect := -1.0
                        End;
                    End;
                Lamp:   With Lmp[Obj] Do
                    Begin
                        VecSub(Loc, Pt, Temp);
                        b := VecDot(Dir, Temp) * - 2.0;
                        c := VecDot(Temp, Temp) - RadSqr;
                        disc := Sqr(b) - 4.0 * c;
                        { a = 1 since VecDot(Dir, Dir) = 1.0 }
                        If Not(disc > 0) Then
                            Intersect := -1.0
                        Else
                            Begin
                                sroot := Sqrt(disc);
                                t1 := (-b-sroot) * 0.5; { 0.5 = 1 / 2a }
                                t2 := (-b+sroot) * 0.5;
                                QuadraticIntersectionCheck
                            End
                        End;
                    End;
                End;
    End;
```


THE RAY TRACING PROGRAM

```
Triangle: With Tri[Obj] Do
  Begin
    dot := VecDot(Norm, Dir);
    If (Abs(dot) < Small) Then
      Intersect := -1.0
    Else
      Begin
        pos1 := NdotLoc;
        pos2 := VecDot(Norm, Pt);
        t := (pos1 - pos2) / dot;
        GetIntrPt(Pt, Dir, t, IntrPoint);
        VecSub(IntrPoint, Loc, delta);
        If ((Abs(Norm[0]) > Abs(Norm[1])) And
            (Abs(Norm[0]) > Abs(Norm[2]))) Then
          SetUpTriangle(1,2)
        Else
          Begin
            If (Abs(Norm[1]) >= Abs(Norm[2]))
              Then SetUpTriangle(0,2)
            Else
              SetUpTriangle(0,1)
          End;
        If EvenCrossings(3) Then
          Intersect := -1.0
        Else
          Intersect := t
        End
      End;
  End;

Parallelogram: With Para[Obj] Do
  Begin
    dot := VecDot(Norm, Dir);
    If (Abs(dot) < Small) Then
      Intersect := -1.0
    Else
      Begin
        pos1 := NdotLoc;
        pos2 := VecDot(Norm, Pt);
        t := (pos1 - pos2) / dot;
        GetIntrPt(Pt, Dir, t, IntrPoint);
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
VecSub(IntrPoint, Loc, delta);
If ((Abs(Norm[0]) > Abs(Norm[1])) And
    (Abs(Norm[0]) > Abs(Norm[2]))) Then
    SetUpParallelogram(1,2)
Else
    Begin
        If (Abs(Norm[1]) >= Abs(Norm[2]))
            Then SetUpParallelogram(0,2)
        Else
            SetUpParallelogram(0,1)
    End;
    If EvenCrossings(4) Then
        Intersect := -1.0
    Else
        Intersect := t
End
End;

Circles: With Cir[Obj] Do
    Begin
        dot := VecDot(Norm, Dir);
        If (Abs(dot) < Small) Then
            Intersect := -1.0
        Else
            Begin
                pos1 := NdotLoc;
                pos2 := VecDot(Norm, Pt);
                t := (pos1 - pos2) / dot;
                GetIntrPt(Pt, Dir, t, IntrPoint);
                VecSub(IntrPoint, Loc, delta);
                rad := Sqrt( VecDot(delta, delta) );
                If (rad > Radius) Then
                    Intersect := -1.0
                Else
                    Intersect := t
            End
        End;
    End;

Ring: With Rng[Obj] Do
    Begin
```


THE RAY TRACING PROGRAM

```

dot := VecDot(Norm, Dir);
If (Abs(dot) < Small) Then
    Intersect := -1.0
Else
    Begin
        pos1 := NdotLoc;
        pos2 := VecDot(Norm, Pt);
        t := (pos1 - pos2) / dot;
        GetIntrPt(Pt, Dir, t, IntrPoint);
        VecSub(IntrPoint, Loc, delta);
        rad := Sqrt( VecDot(delta, delta) );
        If ((rad < Rad1) Or (rad > Rad2)) Then
            Intersect := -1.0
        Else
            Intersect := t
    End
End;

Sphere:    With Sphr[Obj] Do
Begin
    VecSub(Loc, Pt, Temp);
    b := VecDot(Dir, Temp) * - 2.0;
    c := VecDot(Temp, Temp) - RadSqr;
    disc := Sqr(b) - 4.0 * c;
    { a = 1 since VecDot(Dir, Dir) = 1.0 }
    If Not(disc > 0) Then
        Intersect := -1.0
    Else
        Begin
            sroot := Sqrt(disc);
            t1 := (-b-sroot) * 0.5; { 0.5 = 1 / 2a }
            t2 := (-b+sroot) * 0.5;
            QuadraticIntersectionCheck
        End
    End;

Quadratic:    QuadraticShapes(Quad[Obj]);

Cone:         QuadraticShapes(Con[Obj]);
Cylinder:    QuadraticShapes(Cyl[Obj])

```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
End  
End;
```

```
Procedure ShootRay(Start, Dir : TDA;  
  Var Shp, Obj : Integer; Var Dist : Real; Var ObjHit : Boolean);  
  
  Var  
    ShapeNum : Integer;  
    ObjectNum : Integer;  
    NewDist : Real;  
  
  Begin  
    Shp := -1;  
    Obj := -1;  
    Dist := -1.0;  
    ObjHit := False;  
  
    For ShapeNum := 0 To MaxShapeType Do  
      For ObjectNum := 1 To ObjCnt[ShapeNum] Do  
        Begin { if the ground is to be displayed }  
          If (ShapeNum = 0) And Not(Gnd.MtlNum = 0) Then  
            NewDist := Intersect(Start, Dir, Ground, 0)  
          Else  
            NewDist := Intersect(Start, Dir, ShapeNum,  
                                  ObjectNum);  
  
          If (NewDist > Small) Then  
            If (Dist = -1.0) Then  
              Begin  
                ObjHit := True;  
                Dist := NewDist;  
                Shp := ShapeNum;  
                Obj := ObjectNum  
              End  
            Else  
              If (NewDist < Dist) Then { Find closest object }  
                Begin  
                  Dist := NewDist;  
                  Shp := ShapeNum;  
                End  
            End  
          End  
        End  
      End  
    End  
  End  
End;
```


THE RAY TRACING PROGRAM

```
Obj := ObjectNum
End
End
End;
```

Calculate Contribution of Local Color Model
at Intersection Point

GetLoclCol - calculate ambient, diffuse reflection and specular
reflection

```
Const
  MaxNoise = 28;

Type
  Matrix = Array[0..MaxNoise, 0..MaxNoise, 0..MaxNoise] Of Word;
  MatrixPtr = ^Matrix;

Var
  NoiseMatrix : MatrixPtr;

Procedure InitNoise;

  Var
    x, y, z : Byte;
    i, j, k : Byte;

  Begin
    New(NoiseMatrix);

    Randomize;
    For x := 0 To MaxNoise Do
      For y := 0 To MaxNoise Do
        For z := 0 To MaxNoise Do
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
Begin
  NoiseMatrix^[x,y,z] := Trunc(Random * 12000);
  If (x = MaxNoise) Then
    i := 0
  Else
    i := x;
  If (y = MaxNoise) Then
    j := 0
  Else
    j := y;
  If (z = MaxNoise) Then
    k := 0
  Else
    k := z;
  NoiseMatrix^[x,y,z] := NoiseMatrix^[i,j,k]
End
End;
```

```
Function Noise(x, y, z : Real) : Integer;
{ harmonic and random functions combined to create a noise function }
{ based on Perlin's (1985) noise function - ideas found in Alan Watt's
-Fundamentals of Three-Dimensional Computer Graphics }
```

```
Var
  ix, iy, iz : Byte;
  ox, oy, oz : Real;
  p000, p001 : Integer;
  p010, p011 : Integer;
  p100, p101 : Integer;
  p110, p111 : Integer;
  p00, p01 : Integer;
  p10, p11 : Integer;
  p0, p1 : Integer;
  d00, d01 : Integer;
  d10, d11 : Integer;
  d0, d1 : Integer;
  d : Integer;
```

```
Begin
  x := Abs(x);
```


THE RAY TRACING PROGRAM

```
y := Abs(y);
z := Abs(z);

{ Find matrix coordinates and fractional offsets }

ix := Trunc(x) Mod MaxNoise;
iy := Trunc(y) Mod MaxNoise;
iz := Trunc(z) Mod MaxNoise;

ox := frac(x);
oy := frac(y);
oz := frac(z);

{ Interpolation to get Noise value at (ix+ox, iy+oy, iz+oz) }

p000 := NoiseMatrix^[ix , iy , iz ];
p001 := NoiseMatrix^[ix , iy , iz+1];
p010 := NoiseMatrix^[ix , iy+1, iz ];
p011 := NoiseMatrix^[ix , iy+1, iz+1];
p100 := NoiseMatrix^[ix+1, iy , iz ];
p101 := NoiseMatrix^[ix+1, iy , iz+1];
p110 := NoiseMatrix^[ix+1, iy+1, iz ];
p111 := NoiseMatrix^[ix+1, iy+1, iz+1];

d00 := p100 - p000;
d01 := p101 - p001;
d10 := p110 - p010;
d11 := p111 - p011;

p00 := Trunc(d00 * ox) + p000;
p01 := Trunc(d01 * ox) + p001;
p10 := Trunc(d10 * ox) + p010;
p11 := Trunc(d11 * ox) + p011;

d0 := p10 - p00;
d1 := p11 - p01;

p0 := Trunc(d0 * oy) + p00;
p1 := Trunc(d1 * oy) + p01;
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
d := p1 - p0;

Noise := Trunc(d * oz) + p0
End;

Procedure MarbleTex(Pt : TDA; Var RGB : TDA);

Var
  i, d      : Real;
  x, y, z   : Real;

Begin
  UnVec(Pt, x, y, z);
  x := x * 0.2;
  d := x + 0.0006 * Noise(x, y * 0.1, z * 0.1);
  d := d * (Trunc(d) Mod 25);
  i := 0.7 + 0.05 * Abs(d - 10 - 20 * Trunc(d * 0.05));
  Vec(i, i, i, RGB)
End;

Procedure WoodTex(Pt : TDA; Var RGB : TDA);

Var
  i, d      : Real;
  x, y, z   : Real;

Begin
  UnVec(Pt, x, y, z);
  x := x * 0.2;
  d := x + 0.0002 * Noise(x, y * 0.1, z * 0.1);
  d := d * (Trunc(d) Mod 25);
  i := 0.7 + 0.05 * Abs(d - 10 - 20 * Trunc(d * 0.05));
  Vec(i, i, i, RGB)
End;

Procedure GetLoc1Col(Shp, Obj : Integer; Dir, IntrPt, SrfNrm : TDA;
  Dist : Real; Var Loc1Col : TDA);
```


THE RAY TRACING PROGRAM

```
Procedure Texture(Tex : Integer; Var Texturing : TDA);
```

```
  Var
```

```
    x, y, z : Integer;
```

```
    lev     : Real;
```

```
  Begin
```

```
    Case Tex Of
```

```
      Checker:  Begin
```

```
        x := Round(Abs(IntrPt[0]) * Tile) Mod 10000;
```

```
        y := Round(Abs(IntrPt[1]) * Tile) Mod 10000;
```

```
        z := Round(Abs(IntrPt[2]) * Tile) Mod 10000;
```

```
        If ((( x + y + z ) Mod 2) = 1) Then
```

```
          VecCopy(Tile1, Texturing)
```

```
        Else
```

```
          VecCopy(Tile2, Texturing)
```

```
      End;
```

```
      Grit:      Begin
```

```
        lev := Random * 0.2 + 0.8;
```

```
        Vec(lev, lev, lev, Texturing)
```

```
      End;
```

```
      Marble:    Begin
```

```
        MarbleTex(IntrPt, Texturing)
```

```
      End;
```

```
      Wood:      Begin
```

```
        WoodTex(IntrPt, Texturing)
```

```
      End;
```

```
      Sheetrock: Begin
```

```
        lev := Random * 0.1 + 0.9;
```

```
        Vec(lev, lev, lev, Texturing)
```

```
      End
```

```
    End
```

```
  End;
```

```
  Var
```

```
    Mt1, Src, Tex : Integer;
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
Amb, Total      : TDA;
ShadShp         : Integer;
ShadObj         : Integer;
ShadDist        : Real;
ObjHit          : Boolean;
Temp            : TDA;
LmpDir          : TDA;
Addition        : TDA;
```

```
Procedure Ambient;
```

```
Begin
  VecElemMult(1.0, Mat1[Mt1].AmbRfl, AmbIntens, Total)
End;
```

```
Procedure UnshadowedSurface;
```

```
Var
  Lamb          : Real;
  Diff, Spec    : TDA;
```

```
Procedure Diffuse;
```

```
Begin
  VecElemMult(Lamb, Mat1[Mt1].DifRfl, Lmp[Src].Intens, Diff)
End;
```

```
Procedure Specular;
```

```
Var
  Temp          : TDA;
  Cone, Glint   : Real;
```

```
Begin
  VecSub(LmpDir, Dir, Temp);
  VecScalMult(0.5, Temp, Temp);      {  $H = (L + V) / 2$  }
  VecNormalize(Temp);
  Cone := VecDot(SrfNrm, Temp);
```


THE RAY TRACING PROGRAM

```
If (Cone < 1E-10) Then Cone := 1E-10;
```

```
Glint := Exp( Mat1[Mt1].Gloss * Log(Cone));
```

```
VecElemMult( Glint, Mat1[Mt1].SpcRfl, Lmp[Src].Intens,  
                                                     Spec)
```

```
End;
```

```
Begin
```

```
Lamb := VecDot(SrfNrm, LmpDir); { N · L }
```

```
If (Lamb <= 0.0) Then
```

```
Begin
```

```
VecNull(Diff);
```

```
VecNull(Spec)
```

```
End
```

```
Else
```

```
Begin
```

```
Diffuse;
```

```
Specular
```

```
End;
```

```
VecAdd(Diff, Spec, Addition)
```

```
End;
```

```
Procedure ShadowFeeler;
```

```
Begin
```

```
ShootRay(IntrPt, LmpDir, ShadShp, ShadObj, ShadDist, ObjHit)
```

```
End;
```

```
Var
```

```
ColorTexture : TDA;
```

```
IntensFactor : Real;
```

```
HitItself : Boolean;
```

```
Begin
```

```
If (Shp = Lamp) Then
```

```
Begin
```

```
IntensFactor := (1.0 - DistEffect) +
```

```
DistEffect * (-VecDot(SrfNrm, Dir) / Sqrt(Dist));
```

```
VecScalMult(IntensFactor, Lmp[Obj].Intens, Loc1Col)
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
End
Else
Begin
  Case Shp Of
    Ground:      With Gnd Do
      Begin
        Mtl := MtlNum;
        Tex := TexNum
      End;
    Triangle:    With Tri[Obj] Do
      Begin
        Mtl := MtlNum;
        Tex := TexNum
      End;
    Parallelogram: With Para[Obj] Do
      Begin
        Mtl := MtlNum;
        Tex := TexNum
      End;
    Circles:      With Cir[Obj] Do
      Begin
        Mtl := MtlNum;
        Tex := TexNum
      End;
    Ring:         With Rng[Obj] Do
      Begin
        Mtl := MtlNum;
        Tex := TexNum
      End;
    Sphere:       With Spshr[Obj] Do
      Begin
        Mtl := MtlNum;
        Tex := TexNum
      End;
```


THE RAY TRACING PROGRAM

```
Quadratic:      With Quad[Obj] Do
    Begin
        Mtl := MtlNum;
        Tex := TexNum
    End;
Cone:           With Con[Obj] Do
    Begin
        Mtl := MtlNum;
        Tex := TexNum
    End;

Cylinder:       With Cyl[Obj] Do
    Begin
        Mtl := MtlNum;
        Tex := TexNum
    End
End;

Ambient;

For Src := 1 To ObjCnt[Lamp] Do { sum all lamp contributions }
    Begin
        VecSub(Lmp[Src].Loc, IntrPt, LmpDir);
        VecNormalize(LmpDir);
        ShadowFeeler;
        { There is no need to check beyond lamp since the
          closest object is returned. If the shadow
          feeler hits the lamp that is in the direction
          of the lamp (itself) then the hit is ignored }
        HitItself := ObjHit And (ShadShp = Lamp) And
                        (ShadObj = Src);

        If Not(ObjHit) Or HitItself Then
            Begin
                UnshadowedSurface;
                VecAdd(Total, Addition, Total)
            End
    End;
End;

If (Tex = Smooth) Then { If a smooth surface, then don't
                        modify it }
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
        VecCopy(Total, Loc1Col)
    Else
        Begin
            Texture(Tex, ColorTexture);
            VecElemMult(1.0, Total, ColorTexture, Loc1Col)
        End
    End
End;
```

```
{
```

Calculate Sky

Sky - blend a sky color from the horizon to the zenith

```
}
```

```
Procedure Sky(Dir : TDA;  Var Col : TDA);
```

```
    Const
```

```
        Small = 1E-03;
```

```
    Var
```

```
        sin2, cos2 : Real;
```

```
        x2, y2, z2 : Real;
```

```
    Begin
```

```
        x2 := Sqr(Dir[0]);
```

```
        y2 := Sqr(Dir[1]);
```

```
        z2 := Sqr(Dir[2]);
```

```
        If (z2 = 0) Then z2 := Small;
```

```
        sin2 := z2 / (x2 + y2 + z2);
```

```
        cos2 := 1.0 - sin2;
```

```
        VecLinComb(cos2, HorCol, sin2, ZenCol, Col)
```

```
    End;
```


THE RAY TRACING PROGRAM

```
{
```

```
Recursive Ray Tracer
```

```
TraceRay - perform recursive ray tracing
```

```
}
```

```
Procedure Comb(A, B, C, D : TDA; Var Col : TDA);
```

```
Var
```

```
T1, T2 : TDA;
```

```
Begin
```

```
VecElemMult(1, A, B, T1); { Local }
```

```
VecElemMult(1, C, D, T2); { Reflected }
```

```
VecAdd(T1, T2, Col)
```

```
End;
```

```
Function WgtMin(TotWgt : TDA) : Boolean;
```

```
Begin
```

```
If (TotWgt[0] <= MinWgt[0]) And  
   (TotWgt[1] <= MinWgt[1]) And  
   (TotWgt[2] <= MinWgt[2]) Then  
   WgtMin := True
```

```
Else
```

```
   WgtMin := False
```

```
End;
```

```
Procedure TraceRay(Start, Dir, TotWgt : TDA; Depth : Integer; Var Col  
                  : TDA);
```

```
Var
```

```
LoclCol, ReflCol : TDA;
```

```
ReflDir, Wgt : TDA;
```

```
IntrPt, SrfNrm : TDA;
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
Shp, Obj      : Integer;  
Dist          : Real;  
ObjHit        : Boolean;
```

```
Function MaterialSpecular : Boolean;
```

```
Var  
    Mtl : Integer;
```

```
Begin  
    Case Shp Of  
        Ground:      With Gnd Do  
                        Mtl := MtlNum;  
  
        Triangle:    With Tri[Obj] Do  
                        Mtl := MtlNum;  
  
        Parallelogram: With Para[Obj] Do  
                        Mtl := MtlNum;  
  
        Circles:      With Cir[Obj] Do  
                        Mtl := MtlNum;  
  
        Ring:         With Rng[Obj] Do  
                        Mtl := MtlNum;  
  
        Sphere:       With Spshr[Obj] Do  
                        Mtl := MtlNum;  
  
        Quadratic:    With Quad[Obj] Do  
                        Mtl := MtlNum;  
  
        Cone:         With Con[Obj] Do  
                        Mtl := MtlNum;  
  
        Cylinder:     With Cyl[Obj] Do  
                        Mtl := MtlNum  
    End;
```

```
If Not(Shp = Lamp) Then
```


THE RAY TRACING PROGRAM

```

If (Matl[Mtl].SpcRfl[0] = 0) And
  (Matl[Mtl].SpcRfl[1] = 0) And
  (Matl[Mtl].SpcRfl[2] = 0) Then
  MaterialSpecular := False
Else
  MaterialSpecular := True
Else
  MaterialSpecular := True
End;

Begin
  ShootRay(Start, Dir, Shp, Obj, Dist, ObjHit);
  If ObjHit Then
    Begin
      GetIntrPt(Start, Dir, Dist, IntrPt);
      GetSrfNrm(Shp, Obj, IntrPt, SrfNrm);

      GetLoclCol(Shp, Obj, Dir, IntrPt, SrfNrm, Dist,
        LoclCol);

      If (Depth = MaxDepth) Or WgtMin(TotWgt) Then
        VecElemMult(1.0, LoclCol, LoclWgt, Col)
      Else
        Begin
          If (Not(Shp = Lamp) Or (Shp = Lamp) And
            ReflectiveLamps) Then
            If MaterialSpecular Then
              Begin
                CalcDirOfSpecRef1Ray(Dir, SrfNrm,
                  ReflDir);
                VecElemMult(1, TotWgt, ReflWgt, Wgt);
                TraceRay(IntrPt, ReflDir, Wgt, Depth+1,
                  ReflCol)
              End
            Else
              VecNull(ReflCol)
            Else
              VecNull(ReflCol);
          Comb(LoclCol, LoclWgt, ReflCol, ReflWgt, Col)
        End
      End
    End
  End

```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
        End
      Else
        Sky(Dir, Col)
      End;

```

```
{
```

Scan Pixel Display

GetEye - calculate the direction of a ray from the eye through
a screen pixel and into the scene

Scan - scan the pixel display with eye-rays

```
}
```

```
Var
```

```
  ViewVec      : TDA;
  CenterX      : Integer;
  CenterY      : Integer;
  ScanXRes     : Word;
  ScanYRes     : Word;
  XPitchDivFocLen : Real;
  YPitchDivFocLen : Real;

```

```
Procedure Precalculation;
```

```
  Var
```

```
    Scale : Real;
```

```
  Begin
```

```
    XPitchDivFocLen := XPitch / FocalLength;
    YPitchDivFocLen := YPitch / FocalLength;
    CenterX := ScanXRes Div 2;
    CenterY := ScanYRes Div 2;
    Scale := CenterX;
    VecScalMult(Scale, ViewDir, ViewVec)
  End;

```


THE RAY TRACING PROGRAM

End;

Procedure GetInitialDir(i, j : Integer; Var Dir : TDA);

Var

x, y : Real;
EyeToPixVec : TDA;

Begin

x := (i - CenterX) * XPitchDivFocLen;
y := (CenterY - j) * YPitchDivFocLen;
VecLinComb(x, U, y, V, EyeToPixVec);
VecAdd(ViewVec, EyeToPixVec, Dir);
VecNormalize(Dir)

End;

Procedure Scan(FileName : Name);

Var

InitialDir : TDA;
Col : TDA;
Colr : TDIA;
Xp, Yp : Integer;

Begin

Precalculation;

Assign(TextDiskFile, FileName);

Rewrite(TextDiskFile);

WriteLn(TextDiskFile, ScanXRes, ' ', ScanYRes);

For Yp := 0 To ScanYRes-1 Do

For Xp := 0 To ScanXRes-1 Do

Begin

GetInitialDir(Xp, Yp, InitialDir);

TraceRay(ObsPos, InitialDir, MaxWgt, 1, Col);

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
        VecScalMultInt(63.0, Col, Colr);
        Plot(Xp, Yp, (Colr[0] + Colr[1] + Colr[2]) Div 3);
        Write(TextDiskFile, Chr(Colr[0]), Chr(Colr[1]),
                                                    Chr(Colr[2]))
    End;

    Close(TextDiskFile)
End;

Var
    FileName : Name;

Procedure GetFileName;

    Var
        x, y : Byte;

    Begin
        WriteLn;
        Write('Enter File Name => ');
        x := WhereX;
        y := WhereY;
        ReadLn(FileName);
        If (FileName = '') Then
            Begin
                FileName := 'Example1';
                GotoXY(x,y);
                WriteLn(FileName)
            End;
        FileName := FileName + '.RT';
        WriteLn;
        WriteLn
    End;

{


Main Program


}
```


THE RAY TRACING PROGRAM

```
Var
  PalArray : PaletteRegister;

Begin
  InitNoise;           { Noise Function for Marble Textures }

  ClearMemory;

  ClrScr;

  GetFileName;

  GetData(FileName);

  ScanXRes := XRes;
  ScanYRes := YRes;

  If XRes = 1024 Then
    InitGraphics(56)
  Else
    InitGraphics(19);
  InitPalette1(PalArray);
  SetPalette(PalArray);

  Scan(OutputFile);

  Dispose(NoiseMatrix);
  NoiseMatrix := nil;

  ExitGraphics
End.
```

Figure 19-1. Listing of the Ray Tracing Program

Loading an *.RT* File

In order to observe how an *.RT* file is loaded, you need to refer to the section of the program listing titled "Load a *.RT File", which describes the procedure *GetData*. At the beginning of this procedure are a number of other procedures, each of which handles the loading of a particular type of data. A few of these will be discussed now, and the rest covered in the *GetData* procedure section. The first one to examine is the *LoadTDA* procedure. This procedure first reads a line from the text file into a buffer. It creates a vector by iterating a loop three times which converts the eight characters beginning at position $14 + i*9$ (where i is the iteration index) into a real number (which is stored in the indexed member of a vector). Note that this means the text descriptor beginning the file line is completely ignored and it requires the exact spacing of each of the three numbers that make up the vector. No deviation is allowed, so be careful how you space your *.RT* files. The next procedure, *LoadReal*, also reads a line from the text file into a buffer. It creates a real number by converting the eight characters beginning at position 14 into a real number. Again the text descriptor is ignored and the positioning of the numerical data in the file is critical. The next procedure, *LoadInteger*, also reads a line from the text file into a buffer. It then creates an integer by converting the eight characters beginning at position 14 into an integer. Again the text descriptor is ignored and the positioning of the numerical data in the file is critical. The next procedure, *LoadText*, also reads a line from the text file into a buffer. It then makes a text string out of the 30 characters beginning at location 14 of the file line. Again the text descriptor is ignored and the positioning of the string data in the file is critical. For now, jump down to the listing of the *GetData* procedure itself. It begins by opening and resetting the designated data file and begins a *Repeat* loop which is iterated until the end of file is encountered. At each iteration, the first thing that the procedure does is to read a line from the file into a buffer. The rest of the loop is a series of *If* statements, one for each type of information encountered in the buffer as a heading for a section of the file. The first is *STATS*. If this is encountered, the procedure reads a line which is the output file name. It then reads two lines of integers which are the horizontal and vertical resolution of the desired display. It then reads *XPitch* and *YPitch*, described above. If the buffer contained *Environment*, the procedure reads four vectors, *LoclWgt*,

ReflWgt, *MinWgt*, and *MaxWgt*. It then reads the integer *MaxDepth*. These were all described above. If the buffer contains *OBSERVER*, the procedure reads the real variable *FocalLength*, followed by the vector *ObsPos*, followed by the real-number angles *ObsRotate* and *ObsTilt*. It then calls the procedure *GetViewDir*. This procedure computes the angles Φ and Θ (in radians) and then the View vector. It then computes the transformation vectors *U* and *V*. Returning now to the *GetData* procedure, if the buffer contains *SKY* the procedure reads the vectors for the horizon and zenith colors. If the buffer contains *CHECK*, the procedure reads in the two color vectors for the checkerboard and the real number that is the length of the side of each square. If the buffer contains *MATERIAL*, the procedure first increments the count of material types. It then loads the proper members of the *Matl* array at the new count location; the name of the material type and texture, the vectors representing the ambient, diffuse, and specular colors of the material, and the real number representing its glossiness power. If the buffer contains *AMBIENTLIGHT*, the procedure loads the vector that contains the ambient light color. If the buffer contains *GROUND*, the procedure first loads the material name for the ground. It then calls *GetMatlNum*, which searches through the list of materials until it encounters one with the specified name, and assigns its number to the proper member of the ground array. It then calls *GetTexNum*, which assigns a texture number corresponding to the texture specified for the designated material. If the buffer contains *LAMP*, the procedure increments the number of lamps and then loads the variables needed to define a lamp into the proper members of the lamp array. If the buffer contains *TRIANGLE*, the procedure increments the number of triangles and then loads the variables needed to define a triangle into the proper members of the triangle array. It then performs the vector cross-product of the two vectors that define the triangle to obtain the normal to the triangle surface. It computes and stores the dot product of the normal and the location vector of the triangle and calls *GetMatlNum* and *GetTexNum* to obtain the material and texture numbers, described above. If the buffer contains *PARALLELOGRAM*, the procedure increments the number of parallelograms and then loads the variables needed to define a parallelogram into the proper members of the parallelogram array. It then performs the vector cross-product of the two vectors that define the parallelogram to obtain the normal to the parallelogram surface. It then computes and

stores the dot product of the normal and the location vector of the parallelogram. It then calls *GetMatlNum* and *GetTexNum* to obtain the material and texture numbers as described above. If the buffer contains *CIRCLE*, the procedure increments the number of circles and then loads the variables needed to define a circle into the proper members of the circles array. It then performs the vector cross-product of the two vectors that define the circle to obtain the normal to the circle surface and computes and stores the dot product of the normal and the location vector of the circle. It then calls *GetMatlNum* and *GetTexNum* to obtain the material and texture numbers, described above. If the buffer contains *RING*, the procedure increments the number of rings and then loads the variables needed to define a ring into the proper members of the rings array. It then performs the vector cross product of the two vectors that define the ring to obtain the normal to the ring surface. It then computes and stores the dot product of the normal and the location vector of the ring. It then calls *GetMatlNum* and *GetTexNum* to obtain the material and texture numbers as described above. If the buffer contains *SPHERE*, the procedure increments the number of spheres and then loads the variables needed to define a ring into the proper members of the spheres array. It then calls *GetMatlNum* and *GetTexNum* to obtain the material and texture numbers as described above. If the buffer contains *QUADRATIC*, the procedure increments the number of quadratics, loads the variables needed to define a quadratic into the proper members of the quadratic array. The procedure also calls *OrientQuadratic*, which performs the necessary mathematical operations to tilt the quadratic if it is not to be oriented in the y direction, which is into the screen. It then calls *GetMatlNum* and *GetTexNum* to obtain the material and texture numbers as described above. If the buffer contains *CONE*, the cone is actually a quadratic, but is specified in terms of the radius of its base circle and height of its apex rather than in terms of the quadratic equation. The procedure thus loads these parameters and automatically computes the necessary coefficients for the quadratic equation. It increments the number of quadratics and stores the variables needed to define the cone quadratic into the proper members of the quadratic array. It then calls *GetMatlNum* and *GetTexNum* to obtain the material and texture numbers, described above. If the buffer contains *CYLINDER*, the cylinder is actually a quadratic, but is specified in terms of the radius of its circle and its height rather than in terms of the

quadratic equation. The procedure thus loads these parameters and then automatically computes the necessary coefficients for the quadratic equation. It then increments the number of quadratics and then stores the variables needed to define the cylinder quadratic into the proper members of the quadratic array. It then calls *GetMatlNum* and *GetTexNum* to obtain the material and texture numbers as described above. If the buffer contains *BOX*, the procedure reads the location vector and the three vectors needed to define the shape and position of the box. It then calls *GetMatlNum* and *GetTexNum* to obtain the material and texture numbers as described above. Then, the procedure determines the characteristics of the six parallelograms that are required to define the sides of the box and performs the same operations for each one that it ordinarily performs for a parallelogram.

If the buffer contains *PYRAMID*, the procedure reads the location vector, the two vectors and the height needed to define the shape and position of the pyramid. It then calls *GetMatlNum* and *GetTexNum* to obtain the material and texture numbers as described above. Then, the procedure determines the characteristics of the four triangles that are required to define the sides of the pyramid and performs the same operations for each one that it ordinarily performs for a triangle. When the loop iterates until the end of file is reached, the loop terminates, the file is closed and the procedure returns to the calling entity.

Initializing the Noise Function

To modify the color data, apply a noise function to produce textures similar to wood and marble. The technique that follows is described by Alan Watt in his book, *Fundamentals of Three-Dimensional Computer Graphics* (Addison-Wesley, 1989). Begin with a 29 x 29 x 29 matrix. Next, the procedure uses three, nested *For* loops to fill every member of this array with a random number between 0 and 12000. This is in the first statement under the innermost *For* loop. The remainder of the procedure is designed so that whenever one of three indices of the array reaches its maximum value, the array cell fills with the same number contained in the corresponding cell where the index value is zero. This assures that the end and beginning of the array are the same, and cycles without undue effects.

Scanning the Scene

Now let's look at *Scan*, which scans the pixel display: It calls the procedure *Precalculation*. Remember that *YPitch* must be modified for 320 x 200 display in the *.RT* files. (*XPitch* = *YPitch* = 1 for the 1024 x 768 mode.) This is necessary because the horizontal- and vertical-pixel dimensions are the same in the high resolution mode, but not in the low-resolution mode. Hence, if the picture proportions are correct for the high-resolution mode, you have to perform a scaling in one dimension for the picture to have the same proportions in the lower-resolution mode. The procedure computes the scaling factors *XPitchDivFocLen* and *YPitchDivFocLen*. It determines the center of the screen and computes the vector *ViewVec*, which is the vector from the observer position to the center of the screen. Getting back to the *Scan* procedure, the next operation is to open and prepare to rewrite the designated output file. The first things written to the file are the *x* and *y* resolution values for the display. Next the procedure starts two nested *For* loops which iterate once for each pixel location on the display. For each iteration, the procedure first calls *GetInitialDir*. This procedure performs the necessary mathematical operations to compute a normalized vector *InitialDir* in the direction from the observer, to the pixel on the screen that is being processed. The *Scan* procedure calls *TraceRay*, which will project this vector until it hits an object in the scene and then report back the resulting color for the pixel. Next, the colors, which are values in the range from 0 to 1 are scaled to be in the range of 0 to 63, which is the color information you want to store in the output file. First, sum the colors together and divide by 63 to get a gray scale value, which is immediately plotted on the screen. Then the color information is written to the output file. When the loops are through, this has been done for every pixel on the screen. You'll have a complete gray scale image of the scene on the screen and a complete file of color data stored on the disk.

Tracing the Ray

The function *TraceRay* performs the actual chore of tracing the ray from the screen to the objects in the scene. The procedure calls the procedure *ShootRay*. This procedure makes use of two nested *For* loops to examine every type of shape and every object in the data base. For each object, it calls *Intersect* to determine the

distance at which the current ray from the observer intersects the object. If this distance is greater than a minimum and the distance parameter is still at its initial value, a set of parameters indicates the shape number and object number of the object being intersected, as well as the distance to intersection, and indicates that an intersection has occurred. If the distance parameter is not at its initial value, the above parameters are reset only if the intersect distance for the current object is less than the distance stored in the parameter. Thus when the loops have completed, the parameters contain information for the object for which the nearest intersection occurred.

Now let's look at *Intersect* to see how it works. You are now working backwards, and have already started at the observer's eye and traced a ray of light back to the screen. You have seen that you test each object for ray-intersect at any point on the object's surface. The *Intersect* procedure does this. It is essentially a big *Case* statement, with a separate case to be run for each different shape encountered. Now determine at some point in your computations exactly where on an object's surface the ray may be hitting. The math for determining the intersect point is different for each shape. Simplify this as much as possible by converting the ray-direction vector to the local coordinate system of the shape being tested and then use the equation of the ray in a parametric form. In particular, the equations for the coordinates of the ray are:

$$x = x_1 t + x_0 \quad (\text{Equation 19-1})$$

$$y = y_1 t + y_0 \quad (\text{Equation 19-2})$$

$$z = z_1 t + z_0 \quad (\text{Equation 19-3})$$

The coordinates (x_0, y_0, z_0) are the coordinates of the starting point of the ray and the coordinates (x_1, y_1, z_1) represent a unit vector in the direction in which the ray is travelling. The parameter t can be taken as time. Thus at zero time, the ray is at its origin, and as time passes, the ray travels farther along its path. You will be interested in determining the time at which the ray intersects an object. You want to find the shortest time for the ray to hit an object's surface and later convert this to

distance. The point of intersection with a sphere is determined as follows. The equation of a sphere, centered at (0, 0, 0) and having a radius r , is:

$$x^2 + y^2 + z^2 - r^2 = 0 \quad (\text{Equation 19-4})$$

If you substitute the parametric equations of the ray into this equation (using for the ray the same coordinate system whose origin is the center of the sphere) you'll obtain the following quadratic equation:

$$(x_1^2 + y_1^2 + z_1^2) t^2 + 2(x_0x_1 + y_0y_1 + z_0z_1) t + (x_0^2 + y_0^2 + z_0^2) - 1 = 0 \quad (\text{Equation 19-5})$$

Performing a straightforward solution of the quadratic equation, you'll obtain two roots. These roots represent intersections of the ray with the sphere; the smallest one represents the closest intersection.

Now let's consider the intersection of the ray with a quadratic curve. This is similar to the sphere intersection, since the sphere is a degenerate case of the general quadratic curve. The most common quadratic shapes are the cylinder, the elliptic paraboloid, the hyperbolic paraboloid, the elliptic cone, the hyperboloid of one sheet, and the hyperboloid of two sheets. All of these shapes can be represented by the following generalized equation:

$$ax^2 + 2bxy + 2cxz + 2dxw + ey^2 + 2fyz + 2gyw + hz^2 + 2izw + jw^2 = 0 \quad (\text{Equation 19-6})$$

Although the above equation covers every possible kind of quadric curve, it is too complex to be easily handled by this simple ray-tracing program. Fortunately most of the common quadratic surfaces can be treated by a sub-set of this equation, which is:

$$Ax^2 + By^2 + Cz^2 + Ey = D \quad (\text{Equation 19-7})$$

THE RAY TRACING PROGRAM

Using the same technique of substituting the parametric ray equations into the equation of the quadratic and then solving the resulting quadratic equation, you'll end up with either of two equations, depending upon the defined parameters. The first is:

$$t^2(Ax_1^2 + By_1^2 + Cz_1^2) + 2t(Ax_0x_1 + By_0y_1 + Cz_0z_1) + (Ax_0^2 + By_0^2 + Cz_0^2 - D) = 0$$

(Equation 19-8)

the second possible equation is:

$$t^2(Ax_1^2 + Cz_1^2) + 2t(Ax_0x_1 - Ey_1 + Cz_0z_1) + (Ax_0^2 - Ey_0 + Cz_0^2) = 0$$

(Equation 19-9)

Next, let's consider an intersection with a ring. This is the first of three surfaces that are actually sections of a plane. First determine if the ray intersects with the plane containing the ring. The equation for a plane is:

$$ax + by + cz + d = 0$$

(Equation 19-10)

The intersection of the ray with the plane is obtained, as before, by substituting the parametric equations of the ray into the equation of the plane, giving:

$$t = \frac{ax_0 + by_0 + cz_0 + d}{ax_1 + by_1 + cz_1 + d}$$

(Equation 19-11)

This is a general equation, but use a coordinate system centered at the middle of the plane, so that the parameter d will always be zero. As you look at this equation, observe that the vector (a, b, c) is the normal to the plane. Consequently, the denominator of the equation is the dot product of the normal to the plane and the line direction. As for the numerator, it is the dot product of the normal to the plane with the difference between the origin-of-the-plane coordinate system and the origin of

the line. Next, find the vector from the plane coordinate origin to the point of intersection. The length of this vector is the square root of the dot product of the vector and itself. This length is called *rad*. If *rad* falls between the values of the inner radius of the ring and the outer radius of the ring, you'll have a valid intersection with the ring.

Determining the intersection point of the ray with a parallelogram starts in the same way as for the ring. Determine if the ray intersects the plane containing the parallelogram. This is more difficult than determining if the intersection with the plane occurs. You'll need to use the Jordan curve theorem, which states that if you project a line from the intersection point in an arbitrary direction, if the line has an odd number of intersections with the line segments that are the sides of the parallelogram, the intersect point is inside the parallelogram. If there is an even number of intersections, the intersect point is outside the parallelogram. To make things easier, first project both the parallelogram and the intersect point onto a plane defined by two axes of the coordinate system. Decide which of the three coordinate axes should be discarded, namely the one which has the largest value for any of the vertices of the parallelogram. This is the dominant axis. Then perform the projection onto the plane formed by the other two coordinates by simply throwing away the values of the dominant coordinate for each of the vertices of the parallelogram. Next, compute the location of each vertex in the new system, with respect to the parallelogram coordinate system. The first vertex is always at (0, 0) by definition. Take into consideration which was the dominant axis and compute the coordinates of each vertex in terms of a coordinate system centered at the intersect point. The vertices in the new coordinate system are stored in $(gu[n], gv[n])$, where n is the vertex number from 0 to 3. The new coordinate system will be considered to have the axes u and v . The arbitrary direction in which to project the test line is, for simplicity, selected along the u axis. The function *EvenCrossings* checks the test line against the line segment between each pair of adjacent vertices. You can only have an intersection of the test line and the line segment connecting adjacent vertices if $gv[i]$ and $gv[j]$ have opposite signs. (This puts one vertex on one side of the u axis and the other on the opposite side, which assures a crossing of the u axis.) If both $gu[i]$ and $gu[j]$ are positive, then you'll know that the crossing of the u axis is also a crossing

of the test line, and you therefore increment the number of crossings. If one of the vertex u coordinates is positive, and the other is negative, you'll want to compute the value of the u coordinate for the intersection of the line segment with the u axis. If this is positive, you have an intersection with the test line and therefore need to increment the number of crossings; if it is negative, there is no intersection with the test line, so the number of crossings remains unchanged. After completing the test for all adjacent line segment pairs, the function returns *True* if there was an even number of crossings and *False* if there was not. A return of *True* indicates no intersection; a return of *False* indicates that the intersection occurred.

Finding the intersection of the ray with a triangle works in just the same way, except that only three vertices are specified, and therefore only three line segments must be tested.

You're now getting pretty far removed from where you started, so try to get back as quickly as possible by noting that *Intersect* returns the distance at which the ray intercepts the designated object, and that *ShootRay* then completes its loops by identifying the object intersected and the distance from the ray origin to it. This information is returned to *TraceRay*. If there was no intersection with an object, the procedure *Sky* is called. This procedure interpolates between the horizon and zenith sky colors, as appropriate for the position in the sky that the ray is pointing toward, and returns the proper color. If the ray did intersect an object, the procedure *GetIntrPt* is called to establish the coordinates of the point of intersection of the ray and the object. The procedure *GetSrfNrm* is then called to determine the surface normal vector to the object surface at the point of intersection of the ray with the object. The mathematics for obtaining the surface normal is different for each object; what's involved is in the listing for the procedure.

Determining the Color

The next thing that *TraceRay* does is to call the procedure *GetLoclCol*, which returns the local color of the object at the point of intersection. At this point you're going to find that the procedures and functions are rather convoluted so that very careful tracing through the listing is needed to understand all of the necessary interactions. The *GetLoclCol* procedure begins by checking if the object that was

intersected by the ray is a lamp. If so, you won't need to go any further. The color is set to that of the lamp, with its intensity decreased by the square root of the distance from the lamp to the viewer and by the cosine of the angle between the lamp to viewer direction and the surface normal to the lamp surface. If the object intersected by the light ray is not a lamp, the procedure next enters a *Case* statement which determines the numbers for the type of material and texture of the intersected surface. The procedure *Ambient* adds the contribution of ambient light to the total light color vector. The *GetLoclCol* procedure then enters a *For* loop that iterates once for each lamp (light source) that is in the scene, thereby determining the contributions of all the light sources to the lighting on the object at the point where it is intersected by the light ray. Each iteration of the loop begins by establishing a new light ray from the point of intersection on the object surface toward the lamp being processed. Next, the procedure *ShadowFeeler* is called, which really only calls *ShootRay* to determine the closest object this new ray hits. If you do intersect with an object which is not the lamp itself, your initial intersection will be shadowed from this light source, so you won't have any contribution from it to the color vector. If the new light ray doesn't intersect with anything or if it intersects the lamp itself, there is a color contribution from this light source. In this case, the procedure calls *UnshadowedSurface*. This procedure first checks cosine of the angle between the surface normal and the lamp direction vector. If it is negative, the light from this lamp does not contribute to the lighting of the intersection pixel. Otherwise, the procedure first calls *Diffuse*, which computes the diffuse light contribution and uses it to modify the total color. It then calls *Specular*, which determines the specular reflection contribution to the color. This is obtained from the Phong shading model, which basically takes the cosine of the angle between the surface normal and a vector whose direction is the difference between the viewer and lamp directions from the point of intersection, and raises it to the power specified for glossiness. This gives the intense highlights that are particularly characteristic of reflections from spheres. The procedure *UnshadowedSurface* terminates after having included the diffuse and specular contributions to the total color vector. If the texture of the object first intersected is smooth, you have now determined the total color vector and return to *TraceRay*. If the surface has a texture, the procedure calls *Texture* to perturbate the color thus far determined.

Creating a Textured Surface

The procedure *Texture* modifies the color for each point on a surface to produce the appearance of one of the defined textures. It consists of a *Case* statement which permits the proper operations to be performed for whichever defined texture. The first case is *Checker*, which produces a checkerboard pattern. For this case, the intersection point is processed to determine within which colored square of the checkerboard it falls, and the proper color is then returned to modify the local color information to obtain the final pixel color.

The next case is *Grit*, which produces a slightly varied color over the surface, similar to that which might occur for a grassy surface or for dirt. All that happens for this case is that a random number between 0.8 and 1.0 is selected and multiplies each component of the total color vector.

The next case is *Marble*. The procedure *MarbleTex* creates marble texture. It first separates the point of intersection vector into its three components and then calls the Noise procedure to interpolate in the noise table previously created, and obtain the proper noise value for the point in space ($0.2x$, $0.1y$, $0.1z$). This noise value is modified and creates a vector (having the same value for each of its three components) with which to multiply the local color value.

The next case is *Wood*. The procedure *WoodTex* creates wood texture. This is just the same as the procedure for marble except that the contribution of the noise is only one-third of that for marble.

The last case is *Sheetrock*. This is the same as grit except that the random scalar is set between 0.9 and 1.0 instead of 0.8 to 1.0.

Completing the Ray Trace

You are now back in the *TraceRay* procedure with a color for the pixel. If you have reached the maximum depth of recursion (to be explained as follows), or the weighing of the color on this pass is insignificant, accept the current pixel color and terminate the procedure. Otherwise, if the object material has a specular reflection capability, the procedure calls *CalcDirOfSpecReflRay*, which determines the direction of a ray reflected from the intersection point of the original ray and the object. The procedure computes the reflected ray's contribution to color and *TraceRay* is

called recursively (with the depth increased by one) to perform the whole process over again and determine the color contribution for the reflected ray. The reason for having a depth parameter is to end the recursive process. When the depth reaches the maximum, defined depth (usually 5) further recursion ceases. Next *Comb* is called to combine all of the color contributions into a single, final color vector. This completes *TraceRay* and you'll return one level higher in the recursion, or at the top you return to *Scan*.

Displaying a Ray-Traced File

Using the ray-tracing program produces a *.CPR* disk file which contains color information defining a lot more than the 256 colors available with the present crop of VGA cards and monitors. New cards are coming, however, that allow more than 256 colors, so don't despair; your ray-traced files will be ready for use with the new displays. Because of the large amount of information in the ray-traced file, however, the file uses a lot of space; in fact it is probably not possible to fit a high resolution file onto a standard 360K floppy disk. This chapter looks at how to convert from the file generated by the ray-tracer to the best possible VGA display. You can then capture the screen with any number of commercial or shareware programs and save it in a modest-sized file for redisplay. One such commercial program is *Pizazz Plus*, which not only captures any of the display screens you generate in this chapter and saves to disk for redisplay, but also prints the pictures out with reasonably good quality to a number of color or black and white printers.

Creating a VGA Color Display

The file generated by the ray-tracer begins with two integers that specify the *x* and *y* resolution of the display. After that, it consists of a triple for each pixel on the display screen. The triple is three bytes, each containing a number from 0 to 63, one byte for red, one for green, and one for blue. Each triple can represent 262,144 different shades of color. This is adequate to represent the coloring of any scene with more precision than our human eye can distinguish. Unfortunately, the most colors the VGA card displays on the screen at one time is 256. It might seem like a hopeless task to attempt a realistic scene from our original data file with such a limited number

of colors, but it's not as bad as it seems. Most scenes don't even come close to requiring the full range of colors. Typically, a ray-traced scene specifies from 350 to 3500 colors. Many of these colors shade a surface, and so are very close to each other, and often occur in very limited areas. Consequently, using the 256 most frequently occurring colors produces a realistic scene. The program to perform this operation is listed in Figure 20-1. The program begins by clearing the memory areas for *ColorHistogram*, *Frequency*, *Hues*, and the palette. It then clears the screen, ready to paint a picture. It next displays a heading and then calls *GetFileName*. This procedure asks the user to "Enter File Name =>" and reads the file name in from the keyboard. Only the main part of the file name should be entered; the program automatically adds the extension *.CPR*. If you only enter a carriage return, the program will supply the default file name of *Example1*. Next the main program calls the procedure *CollectColorData*. This procedure begins by opening and resetting the selected file. It then reads the *x* and *y* resolution values and displays them on the screen and displays the line, "Collecting Color Data". The procedure enters two nested *For* loops which read every triple from the screen, strip off the least significant bit, combine the triple into a single-color number and increment the value of the histogram array at the index defined by the color number. After all triples have been read, the file is closed. The procedure then begins a *For* loop which looks at every value in the color histogram. All colors that do not occur are ignored; the others have their color number stored in the *Hues* array and their frequency of occurrence stored in the *Frequency* array. The main program then calls *SortColorData*, which first sorts the colors so that the most frequently occurring one is first and the least frequently occurring one last. The 256 VGA color registers are set to the 256 colors that occur most often in the picture. This is all very well, but what about all of the colors that were not in the first 256 in frequency of occurrence? It is instructive to look at the number of occurrences for these remaining colors. If you want to do this for several typical files, you modify the program of Figure 20-1 so that it completes the sort and prints out the values in the array instead of generating the color picture. You will find that many of the colors below the first 256 only occur once, so that if you get them wrong only a single pixel in the picture is affected. Even the ones that most frequently occur, often have only 20 or 30 occurrences, which is not a very big section of a

picture. Compare each of these colors with the 256 selected for display and assign the color to whichever of the 256 is the closest least-squares fit to the actual color. Next, the main program initiates the graphics mode of operation, selecting mode 56 (1024 x 768 pixels by 256 colors) if the x resolution was 1024 and selecting mode 19 (the standard VGA 320 x 200 pixel by 256 color mode) if the x resolution is anything other than 1024. The high resolution display is only guaranteed to work with the STD Powergraph VGA card; other extended VGA cards may have different mode setting or different methods of operation that require you to change the code. The program then calls *MakePalette* which takes the color assignments in the *Hues* array and uses them to set the 256 VGA color registers. The program calls *GetColorHistogramFromFrequency*. This procedure replaces the frequency of occurrence values in the *ColorHistogram* array with the appropriate VGA color register assignments that are stored in the *Frequency* array. The program then calls *Display*. This procedure reopens and resets the selected file, reads the resolution values, and then enters a pair of nested *For* loops which read each triple, synthesize the color number, use it as an index to get the proper VGA color register value, and then plot a point to the screen at the proper location using the selected color register. When the loops are through, the display is complete. The file is then closed and the program terminates.

```
{
```

Histogram VGA Color Processor

```
}
```

```
Uses
```

```
Dos, Crt;
```

```
{ $I Math2.Inc }
```

```
{ $I Graph2.Inc }
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
Var
  Frequency : Array[0..8191] Of Byte;
  Hues      : Array[0..8191] Of Word;

Procedure Sort(First, Last : Word); { Sort of colors by Frequency }

Var
  i, j, k      : Word;
  Pivot, Temp2 : Word;
  Temp        : Byte;

Begin
  If (First < Last-1) Then
    Begin
      i := First;
      j := Last;
      k := (First + Last) Div 2;
      Pivot := (Frequency[i] + Frequency[j] + Frequency[k])
                Div 3;

      While (i < j) Do
        Begin
          While (Frequency[i] > Pivot) Do
            i := i + 1;
          While (Frequency[j] < Pivot) Do
            j := j - 1;
          If (i < j) Then
            Begin
              Temp := Frequency[i];
              Frequency[i] := Frequency[j];
              Frequency[j] := Temp;
              Temp2 := Hues[i];
              Hues[i] := Hues[j];
              Hues[j] := Temp2;
              i := i + 1;
              j := j - 1;
            End
        End;
    End;
  If (j < Last) Then
    Begin
      Sort(First, j);
```


DISPLAYING A RAY-TRACED FILE

```
        Sort(j+1, Last)
    End
End;
If (Frequency[Last] > Frequency[First]) Then
    Begin
        Temp := Frequency[First];
        Frequency[First] := Frequency[Last];
        Frequency[Last] := Temp;
        Temp2 := Hues[First];
        Hues[First] := Hues[Last];
        Hues[Last] := Temp2
    End
End;
```

```
Type
    Name = String[12];
```

```
Var
    FileName : Name;
```

```
Procedure GetFileName;
```

```
Var
    x, y : Byte;
```

```
Begin
    WriteLn;
    Write('Enter File Name => ');
    x := WhereX;
    y := WhereY;
    ReadLn(FileName);
    If (FileName = '') Then
        Begin
            FileName := 'Example1';
            GotoXY(x,y);
            WriteLn(FileName)
        End;
    FileName := FileName + '.CPR';
    WriteLn;
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
        WriteLn
    End;

Var
    PalArray      : PaletteRegister;
    i, j, x, y    : Word;
    LastColor     : Integer;
    ColorNum      : Word;
    ColorHistogram : Array[0..32767] Of Byte;
    r, g, b       : Char;
    rc, gc, bc    : Word;
    TextDiskFile  : Text;

Procedure ClearMem;

Begin
    ClearPalette(PalArray);

    For i := 0 To 32767 Do
        ColorHistogram[i] := 0;

    For i := 0 To 8191 Do
        Frequency[i] := 0;

    For i := 0 To 8191 Do
        Hues[i] := 0
    End;

Procedure CollectColorData;

Begin
    Assign(TextDiskFile, FileName); { Collect Color Data }
    Reset(TextDiskFile);

    ReadLn(TextDiskFile, XRes, YRes);
```


DISPLAYING A RAY-TRACED FILE

```
WriteLn('Image Resolution is ', XRes, ' by ', YRes, ' pixels.');
```

```
WriteLn;
```

```
WriteLn;
```

```
WriteLn('Collecting color data.');
```

```
WriteLn;
```

```
For y := 0 To YRes-1 Do
  For x := 0 To XRes-1 Do
    Begin
      Read(TextDiskFile, r, g, b);
      rc := (ord(r) AND 62);
      gc := (ord(g) AND 62);
      bc := (ord(b) AND 62);
      ColorNum := (rc ShR 1) OR (gc ShL 4) OR (bc ShL 9);
      If (ColorHistogram[ColorNum] < 255) Then
        ColorHistogram[ColorNum] :=
          ColorHistogram[ColorNum]+1
    End;
```

```
Close(TextDiskFile);
```

```
LastColor := -1;
For j := 0 To 32767 Do
  If (ColorHistogram[j] > 0) Then
    Begin
      LastColor := LastColor + 1;
      Hues[LastColor] := j;
      Frequency[LastColor] := ColorHistogram[j]
    End
```

```
End;
```

```
Procedure SortColorData;
```

```
Var
```

```
  d1, Tempd, d2 : Word;
  r1, g1, b1    : Byte;
  r2, g2, b2    : Byte;
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
Begin
  WriteLn('There are ',LastColor,' colors');
  WriteLn;

  WriteLn('Starting sort. ');
  Sort(0, LastColor);
  WriteLn('Sort completed. ');

  WriteLn;
  WriteLn('Modifying extra colors. ');

  For i := 0 To 255 Do
    Frequency[i] := i;

  For i := 256 To LastColor Do
    Begin
      d1 := 32768;
      For j := 0 To 255 Do
        Begin
          r1 := (Hues[i] ShL 1) AND 62;
          g1 := (Hues[i] ShR 4) AND 62;
          b1 := (Hues[i] ShR 9) AND 62;
          r2 := (Hues[j] ShL 1) AND 62;
          g2 := (Hues[j] ShR 4) AND 62;
          b2 := (Hues[j] ShR 9) AND 62;
          Tempd := Sqr(r1 - r2) + Sqr(g1 - g2) + Sqr(b1 - b2);
          If (Tempd < d1) Then
            Begin
              d1 := Tempd;
              d2 := j;
            End
          End;
        Frequency[i] := d2;
      End;
    End;

  End;

Procedure MakePalette(Var PalArray : PaletteRegister);

Begin
  For j := 0 To 255 Do
```


DISPLAYING A RAY-TRACED FILE

```
Begin
    PalArray[j].Red := (Hues[j] ShL 1) AND 62;
    PalArray[j].Grn := (Hues[j] ShR 4) AND 62;
    PalArray[j].Blu := (Hues[j] ShR 9) AND 62
End
End;
```

Procedure GetColorHistogramFromFrequency;

```
Begin
    For j := 0 To LastColor Do
        ColorHistogram[ Hues[j] ] := Frequency[j]
    End;
```

Procedure Display;

```
Begin
    Assign(TextDiskFile, FileName); { Load in the actual file }
    Reset(TextDiskFile);

    ReadLn(TextDiskFile, XRes, YRes);

    For y := 0 To YRes-1 Do
        For x := 0 To XRes-1 Do
            Begin
                Read(TextDiskFile, r, g, b);
                rc := (ord(r) AND 62);
                gc := (ord(g) AND 62);
                bc := (ord(b) AND 62);
                ColorNum := (rc ShR 1) OR (gc ShL 4) OR (bc ShL 9);
                Plot(x, y, ColorHistogram[ColorNum])
            End;
        End;

    Close(TextDiskFile)
End;
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
{
  MAIN PROGRAM
}
Begin
  ClearMem;

  ClrScr;
  WriteLn('Histogram Color Image Processor using Least Squares
                                                Fit');
  WriteLn('for VGA 256 Color Modes by Christopher D. Watkins');
  WriteLn;

  GetFileName;

  CollectColorData;

  SortColorData;

  If XRes = 1024 Then
    InitGraphics(56)
  Else
    InitGraphics(19);

  MakePalette(PalArray);

  SetPalette(PalArray);

  GetColorHistogramFromFrequency;

  Display;

  ExitGraphics
End.
```

Figure 20-1. Listing of the program *Display.Pas* to display a ray-traced file.

APPENDIX A

Solid Modeling Scene Definition Files

Two data files are necessary to produce a scene using the solid-modeling technique. One is a *.SCN* file, which defines the scene characteristics. These files are on the disk available with this book, and listed in this appendix. The other file that is needed for each scene is the *.DAT* data file. There is a Turbo Pascal program for each of the scenes covered by this book, which generates this data file. How to create the Pascal files for new scenes was described in Chapter 9. Before you attempt to create the scene, compile and run the proper program to generate its data file.

CubePlan.Scen = Cube Sitting on a Reflective Plane

TRUE 0 0 350 500

240 18

45 45

TRUE

FALSE

FALSE

Plane.Dat

7

80.0000 80.0000 80.0000

0.0000 0.0000 0.0000

0.0000 0.0000 0.0000

FALSE

FALSE

TRUE

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

Cube.Dat

1

25.0000 25.0000 25.0000

0.0000 0.0000 0.0000

0.0000 0.0000 25.0000

TRUE

FALSE

FALSE

FourCols.Scen = Four Columns with Spheres on top
reflecting in 3 walls

TRUE 0 -18 350 500

242 24

45 45

TRUE

FALSE

TRUE

Cylinder.Dat

5

15.0 15.0 35.0

0.0 0.0 0.0

-50.0 -50.0 35.0

TRUE

TRUE

FALSE

Sphere.Dat

6

15.0 15.0 15.0

0.0 0.0 0.0

-50.0 -50.0 85.0

TRUE

TRUE

FALSE

SOLID MODELING SCENE DEFINITION FILES

Cylinder.Dat

5

15.0 15.0 35.0

0.0 0.0 0.0

50.0 -50.0 35.0

TRUE

TRUE

FALSE

Sphere.Dat

6

15.0 15.0 15.0

0.0 0.0 0.0

50.0 -50.0 85.0

TRUE

TRUE

FALSE

Cylinder.Dat

5

15.0 15.0 35.0

0.0 0.0 0.0

-50.0 50.0 35.0

TRUE

TRUE

FALSE

Sphere.Dat

6

15.0 15.0 15.0

0.0 0.0 0.0

-50.0 50.0 85.0

TRUE

TRUE

FALSE

Cylinder.Dat

5

15.0 15.0 35.0

0.0 0.0 0.0

50.0 50.0 35.0

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

TRUE
TRUE
FALSE

Sphere.Dat

6

15.0 15.0 15.0
0.0 0.0 0.0
50.0 50.0 85.0

TRUE
TRUE
FALSE

Molecule.Scen = Molecular Model of Water (108° between H-bonds)

FALSE 0 0 500 500
270 0
45 45
FALSE
FALSE
FALSE

Sphere.Dat

1

20.0000 20.0000 20.0000
0.0000 0.0000 0.0000
0.0000 0.0000 0.0000
FALSE
FALSE
FALSE

Cylinder.Dat

7

3.0000 3.0000 15.0000
0.0000 0.0000 0.0000
0.0000 0.0000 35.0000
FALSE
FALSE
FALSE

SOLID MODELING SCENE DEFINITION FILES

Cylinder.Dat

7

3.0000 3.0000 15.0000
 -108.0000 0.0000 0.0000
 0.0000 33.2870 -10.8156
 FALSE
 FALSE
 FALSE

Sphere.Dat

4

10.0000 10.0000 10.0000
 0.0000 0.0000 0.0000
 0.0000 0.0000 60.0000
 FALSE
 FALSE
 FALSE

Sphere.Dat

4

10.0000 10.0000 10.0000
 0.0000 0.0000 0.0000
 0.0000 57.0634 -18.5410
 FALSE
 FALSE
 FALSE

PlotEqn1.Scen = Display an Equation Plot

TRUE 0 -18 450 500
 242 22
 45 45
 FALSE
 FALSE
 FALSE

PlotEqn1.Dat

1

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
100.0 100.0 100.0
  0.0   0.0   0.0
  0.0   0.0   0.0
FALSE
FALSE
FALSE
```

PlotEqn2.Scen = Display an Equation Plot

```
TRUE  0  -18  450  500
242  22
  45  45
FALSE
FALSE
FALSE
```

```
PlotEqn2.Dat
1
```

```
100.0 100.0 100.0
  0.0   0.0   0.0
  0.0   0.0   0.0
FALSE
FALSE
FALSE
```

PlotEqn3.Scen = Display an Equation Plot

```
TRUE  0  -18  450  500
242  22
  45  45
FALSE
FALSE
FALSE
```

```
PlotEqn3.Dat
1
```


SOLID MODELING SCENE DEFINITION FILES

```
100.0 100.0 100.0
  0.0   0.0   0.0
  0.0   0.0   0.0
```

```
FALSE
FALSE
FALSE
```

PlotEqn4.Scen = Display an Equation Plot

```
TRUE 0 -18 450 500
    242 22
    45 45
```

```
FALSE
FALSE
FALSE
```

PlotEqn4.Dat

1

```
100.0 100.0 100.0
  0.0   0.0   0.0
  0.0   0.0   0.0
```

```
FALSE
FALSE
FALSE
```

Shapes.Scen = Collection of Shapes

```
TRUE 0 -24 400 500
    242 24
    45 45
```

```
TRUE
FALSE
TRUE
```

Sphere.Dat

1

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
    30.0  30.0  30.0
      0.0   0.0   0.0
      0.0   0.0  30.0
TRUE
TRUE
FALSE
```

Toroid.Dat

```
4
    30.0  30.0  30.0
      0.0  90.0   0.0
     -60.0 60.0  30.0
TRUE
TRUE
FALSE
```

Cylinder.Dat

```
2
    15.0  15.0  40.0
      0.0   0.0   0.0
     70.0 -70.0  40.0
TRUE
TRUE
FALSE
```

Cone.Dat

```
6
    25.0  25.0  50.0
      0.0   0.0   0.0
      0.0  -65.0 50.0
TRUE
TRUE
FALSE
```

Cube.Dat

```
5
    15.0  15.0  15.0
      0.0   0.0   0.0
     -80.0 10.0  15.0
TRUE
```


SOLID MODELING SCENE DEFINITION FILES

TRUE

FALSE

HmSphere.Dat

3

15.0 15.0 15.0

0.0 0.0 0.0

-80.0 -30.0 55.0

TRUE

TRUE

FALSE

Pyramid.Dat

7

10.0 10.0 10.0

0.0 0.0 0.0

0.0 70.0 10.0

TRUE

TRUE

FALSE

So10fRev.Scen = 6 Solids of Revolution on a Mirrored Grid

TRUE 0 -20 425 500

245 18

45 45

TRUE

FALSE

FALSE

Grid.Dat

7

100.0000 100.0000 100.0000

0.0000 0.0000 0.0000

0.0000 0.0000 0.0000

FALSE

FALSE

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

TRUE

LgBeads.Dat

1

40.0000 40.0000 40.0000

0.0000 0.0000 0.0000

-60.0000 -60.0000 40.0000

TRUE

TRUE

FALSE

Beads.Dat

2

40.0000 40.0000 40.0000

0.0000 0.0000 0.0000

-60.0000 0.0000 40.0000

TRUE

TRUE

FALSE

Funnels.Dat

3

40.0000 40.0000 40.0000

0.0000 0.0000 0.0000

-60.0000 60.0000 40.0000

TRUE

TRUE

FALSE

MechPart.Dat

7

40.0000 40.0000 40.0000

0.0000 0.0000 0.0000

60.0000 -60.0000 40.0000

TRUE

TRUE

FALSE

Rocket.Dat

5

SOLID MODELING SCENE DEFINITION FILES

```
40.0000 40.0000 40.0000
0.0000 0.0000 0.0000
60.0000 0.0000 40.0000
TRUE
TRUE
FALSE
```

ChesPawn.Dat

```
6
40.0000 40.0000 40.0000
0.0000 0.0000 0.0000
60.0000 60.0000 40.0000
TRUE
TRUE
FALSE
```

SphrPlan.Scen = Sphere sitting on a Reflective Plane

```
TRUE 0 0 350 500
240 18
45 45
TRUE
FALSE
FALSE
```

Plane.Dat

```
7
80.0000 80.0000 80.0000
0.0000 0.0000 0.0000
0.0000 0.0000 0.0000
FALSE
FALSE
TRUE
```

Sphere.Dat

```
1
25.0000 25.0000 25.0000
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
0.0000 0.0000 0.0000
0.0000 0.0000 25.0000
TRUE
FALSE
FALSE
```

Sphrwall.Scen = Sphere Reflected in 3 Walls

```
TRUE 0 -18 350 500
242 24
45 45
FALSE
FALSE
FALSE
```

Grid.Dat

7

```
50.0 50.0 50.0
0.0 0.0 0.0
50.0 50.0 0.0
```

```
FALSE
FALSE
TRUE
```

Grid.Dat

7

```
50.0 50.0 50.0
0.0 90.0 0.0
0.0 50.0 50.0
```

```
FALSE
FALSE
TRUE
```

Grid.Dat

7

```
50.0 50.0 50.0
0.0 90.0 90.0
```


SOLID MODELING SCENE DEFINITION FILES

50.0 0.0 50.0

FALSE

FALSE

TRUE

Sphere.Dat

1

25.0 25.0 25.0

0.0 0.0 0.0

30.0 30.0 30.0

TRUE

TRUE

FALSE

StakTors.Scen = Stacked Toroids

TRUE 0 0 350 500

242 24

45 45

TRUE

FALSE

FALSE

Toroid.Dat

7

80.0 80.0 80.0

0.0 0.0 0.0

0.0 0.0 -20.0

TRUE

TRUE

FALSE

Toroid.Dat

1

50.0 50.0 50.0

0.0 0.0 0.0

0.0 0.0 10.0

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

TRUE
TRUE
FALSE

Toroid.Dat

4

30.0	30.0	30.0
0.0	0.0	0.0
0.0	0.0	30.0

TRUE
TRUE
FALSE

Well.Scen = Well with various shapes

TRUE 0 -22 350 500
242 16
45 45

TRUE
FALSE
TRUE

Cylinder.Dat

1

50.0	50.0	3.0
0.0	0.0	0.0
0.0	0.0	3.0

TRUE
TRUE
FALSE

Cylinder.Dat

3

40.0	40.0	2.0
0.0	0.0	0.0
0.0	0.0	5.0

TRUE

SOLID MODELING SCENE DEFINITION FILES

TRUE
FALSE

Cylinder.Dat

1

3.0	3.0	30.0
0.0	0.0	0.0
0.0	35.0	35.0

TRUE
TRUE
FALSE

Cylinder.Dat

1

3.0	3.0	30.0
0.0	0.0	0.0
-25.0	25.0	35.0

TRUE
TRUE
FALSE

Cylinder.Dat

1

3.0	3.0	30.0
0.0	0.0	0.0
-35.0	0.0	35.0

TRUE
TRUE
FALSE

Cylinder.Dat

1

3.0	3.0	30.0
0.0	0.0	0.0
-25.0	-25.0	35.0

TRUE
TRUE
FALSE

Cylinder.Dat

1

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

```
    3.0    3.0    30.0
    0.0    0.0    0.0
    0.0   -35.0   35.0
TRUE
TRUE
FALSE
```

Cylinder.Dat

1

```
    3.0    3.0    30.0
    0.0    0.0    0.0
    25.0   -25.0   35.0
TRUE
TRUE
FALSE
```

Cylinder.Dat

1

```
    3.0    3.0    30.0
    0.0    0.0    0.0
    35.0    0.0    35.0
TRUE
TRUE
FALSE
```

Cylinder.Dat

1

```
    3.0    3.0    30.0
    0.0    0.0    0.0
    25.0   25.0   35.0
TRUE
TRUE
FALSE
```

Cube.Dat

5

```
    15.0   15.0   15.0
    0.0    0.0    0.0
   -65.0   65.0   15.0
TRUE
```


SOLID MODELING SCENE DEFINITION FILES

TRUE
FALSE

Pyramid.Dat

7

25.0	25.0	35.0
0.0	0.0	0.0
65.0	-65.0	35.0

TRUE
TRUE
FALSE

Pyramid.Dat

5

6.0	6.0	12.0
0.0	0.0	0.0
65.0	65.0	12.0

TRUE
TRUE
FALSE

Toroid.Dat

4

35.0	35.0	35.0
0.0	0.0	0.0
-70.0	-70.0	20.0

TRUE
TRUE
FALSE

Sphere.Dat

2

15.0	15.0	15.0
0.0	0.0	0.0
0.0	0.0	35.0

TRUE
TRUE
FALSE

Cone.Dat

6

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

15.0	15.0	35.0
0.0	0.0	0.0
-80.0	0.0	35.0

TRUE
TRUE
FALSE

HmSphere.Dat

1

45.0	45.0	20.0
0.0	0.0	0.0
0.0	0.0	65.0

TRUE
TRUE
FALSE

APPENDIX B

Three Particle Orbits

The parameters that were listed in the program *Orbit-3P.Pas* in Chapter 3 provide an interesting, although somewhat dull, set of orbits, since the orbits are very well behaved, much like the sun and its planets. However, there are other very interesting orbital parameters which cause particles to orbit around each other for awhile and then make violent changes to new orbiting paths. If you would like to explore these, the table on the next page gives parameters for six different orbits that you can try.

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

m1	1	1	1	1	1	1
m2	10	10	10	10	10	10
m3	3	3	3	3	3	3
x1	-60	-40	-40	-40	40	40
y1	0	0	0	0	0	0
x2	0	0	0	0	0	0
y2	0	0	0	0	0	0
x3	90	80	70	90	70	70
y3	0	0	0	0	0	0
vx1	.1000	.1000	.1000	.1010	.1000	.1000
vy1	.1500	.2500	.2500	.2500	.2500	.2500
vx2	.0010	.0010	.0010	.0010	-.0010	-.0010
vy2	.0010	.0010	.0010	.0010	.0010	.0010
vx3	-.0500	-.0500	-.0500	-.0200	-.1010	-.0500
vy3	-.1500	-.1500	-.1500	-.1010	-.1500	-.1500

APPENDIX C

Materials Used in Ray-Tracing

The following file shows the materials developed thus far for use in ray-tracing.

Materials found in *.RT files

```
MATERIAL
  TYPE      = MIRROR
  TEXTURE   = SMOOTH
  AMBRFL    = 0.000 0.000 0.000
  DIFRFL    = 0.700 0.700 0.700
  SPCRFL    = 0.300 0.300 0.300
  GLOSS     = 50.000
```

```
MATERIAL
  TYPE      = DULLMIRROR
  TEXTURE   = SMOOTH
  AMBRFL    = 0.100 0.100 0.100
  DIFRFL    = 0.100 0.100 0.100
  SPCRFL    = 0.100 0.100 0.100
  GLOSS     = 10.000
```

```
MATERIAL
  TYPE      = REDMIRROR
  TEXTURE   = SMOOTH
  AMBRFL    = 0.000 0.000 0.000
```


ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

DIFRFL = 0.700 0.000 0.000
SPCRFL = 0.300 0.000 0.000
GLOSS = 50.000

MATERIAL

TYPE = GREENMIRROR
TEXTURE = SMOOTH
AMBRFL = 0.000 0.000 0.000
DIFRFL = 0.000 0.700 0.000
SPCRFL = 0.000 0.300 0.000
GLOSS = 50.000

MATERIAL

TYPE = BLUEMIRROR
TEXTURE = SMOOTH
AMBRFL = 0.000 0.000 0.000
DIFRFL = 0.000 0.000 0.700
SPCRFL = 0.000 0.000 0.300
GLOSS = 50.000

MATERIAL

TYPE = CHROME
TEXTURE = SMOOTH
AMBRFL = 0.000 0.000 0.000
DIFRFL = 0.250 0.250 0.250
SPCRFL = 0.750 0.750 0.750
GLOSS = 40.000

MATERIAL

TYPE = BRASS
TEXTURE = SMOOTH
AMBRFL = 0.100 0.100 0.100
DIFRFL = 0.300 0.300 0.300
SPCRFL = 0.600 0.500 0.250
GLOSS = 30.000

MATERIALS USED IN RAY TRACING

MATERIAL

TYPE = PLASTICTILE
 TEXTURE = CHECKER
 AMBRFL = 0.100 0.100 0.100
 DIFRFL = 0.700 0.700 0.700
 SPCRFL = 0.200 0.200 0.200
 GLOSS = 4.000

CHECK

TILE1 = 1.000 0.000 0.000
 TILE2 = 0.200 0.200 0.200
 TILE = 0.02

MATERIAL

TYPE = ITALIANMARBLE
 TEXTURE = MARBLE
 AMBRFL = 0.100 0.100 0.100
 DIFRFL = 0.700 0.700 0.700
 SPCRFL = 0.200 0.200 0.200
 GLOSS = 10.000

MATERIAL

TYPE = OAKWOOD
 TEXTURE = WOOD
 AMBRFL = 0.200 0.200 0.200
 DIFRFL = 0.700 0.400 0.300
 SPCRFL = 0.100 0.100 0.100
 GLOSS = 5.000

MATERIAL

TYPE = MATTEWALL
 TEXTURE = SHEETROCK
 AMBRFL = 0.500 0.500 0.500
 DIFRFL = 0.500 0.500 0.500
 SPCRFL = 0.000 0.000 0.000
 GLOSS = 0.000

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

MATERIAL

TYPE = GRASS
TEXTURE = GRIT
AMBRFL = 0.200 0.200 0.200
DIFRFL = 0.600 0.800 0.250
SPCRFL = 0.000 0.000 0.000
GLOSS = 0.000

MATERIAL

TYPE = DULLCEMENT
TEXTURE = GRIT
AMBRFL = 0.200 0.200 0.200
DIFRFL = 0.300 0.300 0.300
SPCRFL = 0.000 0.000 0.000
GLOSS = 0.000

MATERIAL

TYPE = CEMENT
TEXTURE = GRIT
AMBRFL = 0.200 0.200 0.200
DIFRFL = 0.800 0.800 0.800
SPCRFL = 0.000 0.000 0.000
GLOSS = 0.000

MATERIAL

TYPE = LIGHTCEMENT
TEXTURE = SHEETROCK
AMBRFL = 0.100 0.100 0.100
DIFRFL = 0.900 0.900 0.900
SPCRFL = 0.000 0.000 0.000
GLOSS = 0.000

MATERIAL

TYPE = WATER
TEXTURE = GRIT
AMBRFL = 0.100 0.100 0.300

MATERIALS USED IN RAY TRACING

DIFRFL = 0.100 0.300 0.300
 SPCRFL = 0.100 0.100 0.400
 GLOSS = 3.000

MATERIAL

TYPE = LAKEWATER
 TEXTURE = GRIT
 AMBRFL = 0.100 0.100 0.200
 DIFRFL = 0.100 0.200 0.200
 SPCRFL = 0.100 0.100 0.300
 GLOSS = 2.000

MATERIAL

TYPE = BLACKANODIZED
 TEXTURE = SMOOTH
 AMBRFL = 1.000 1.000 1.000
 DIFRFL = 0.000 0.000 0.000
 SPCRFL = 0.000 0.000 0.000
 GLOSS = 0.000

MATERIAL

TYPE = MILKWHITEGLASS
 TEXTURE = SMOOTH
 AMBRFL = 0.500 0.500 0.500
 DIFRFL = 0.400 0.400 0.400
 SPCRFL = 0.100 0.100 0.100
 GLOSS = 4.000

Bibliography

Becker, K.H., and Dorfler, M., *Computergraphische Experimente mit Pascal*, Vieweg, Braunschweig, 1986.

Bouville, C., "Bounding Ellipsoids for Ray-Fractal Intersection." *SIGGRAPH '85*, Vol. 19, No. 3, (1985) pp. 45–52.

Carpenter, L. "Computer Rendering of Fractal Curves and Surfaces." *Computer Graphics*, (1980), pp. 109 ff.

Coquillast, S. and Gangnet, "Shaded Display of Digital Maps." *IEEE Computer Graphics and Applications*, Vol. 4, No. 7, (1984).

Demko, S., Hodges, L., and Naylor, B., "Construction of Fractal Objects with Iterated Function Systems." *SIGGRAPH '85*, Vol. 19, No. 3, (1985) pp. 271–278.

Dewdney, A.K., "Computer Recreations: Exploring the Mandelbrot Set." *Computer Graphics*, Vol. 20, No. 4, (1985), pp. 16 ff.

Dewdney, A.K., "Computer Recreations: Beauty and Profundity: The Mandelbrot Set and a Flock of Its Cousins Called Julia Sets." *Scientific American*, (November 1987), pp. 140–144.

Escher, M.C., *The World of M.C. Escher*, New York: H.N. Abrams, 1971.

Feigenbaum, M.J., "Quantitative Universality for a Class of Non-Linear Transformations." *Journal of Statistical Physics*, Vol. 19, No. 1, (January 1978), pp. 25–52.

Fishman, B. and Schachter, B., "Computer Display of Height Fields." *Computers and Graphics*, No. 5, (1980), pp. 53–60.

Fogg, L., "Drawing the Mandelbrot and Julia Sets." *MicroCornucopia*, No. 39, (January–February 1988), pp. 6–9.

- Glasner, Andrew S., *An Introduction to Ray Tracing*, Academic Press, Ltd., 1989.
- Henon, M., "A Two-Dimensional Mapping with a Strange Attractor." *Communication Math-Physics*, No. 50, (1976), pp. 69–77.
- Mandelbrot, B., *The Fractal Geometry of Nature*, New York: W.H. Freeman and Company, 1983.
- Mastin, G.A., Watterberg, P.A., and Mareda, J.F., "Fourier Synthesis of Ocean Scenes." *IEEE Computer Graphics and Applications*, (March 1987), pp. 16–24.
- Norton, A., "Generation and Display of Geometric Fractals in 3-D." *Computer Graphics*, Vol. 16, No. 3, (July 1982), pp. 61–67.
- Norton, A., "Julia Sets in the Quaternions." IBM T.J. Watson Research Center, Yorktown Heights, New York.
- Peitgen, H.O., and Richter, P.H., *The Beauty of Fractals*, Berlin: Springer-Verlag, 1986.
- Peitgen, H.O. and Saupe, D., *The Science of Fractal Images*, Berlin: Springer-Verlag, 1988.
- Pickover, C.A., "A Note on Rendering 3-D Strange Attractors." *Computers and Graphics*, Vol. 12, No. 2, (1988), pp. 263–267.
- Pickover, C.A. "Chaotic Behavior of the Transcendental Mapping ($z \rightarrow \cosh(z) + u$)." *The Visual Computer: An International Journal of Computer Graphics*, No. 4, pp. 243–246.
- Pickover, C.A., "Mathematics and Beauty VII: Visualization of Quaternion Slices." *Image and Vision Computing*, Vol. 6, No. 4, (November 1988), pp. 235–237.
- Sorensen, P., "Fractals." *Byte*, (September 1984), pp. 157–171.
- Watt, Alan, *Fundamentals of Three-Dimensional Computer Graphics*, Addison-Wesley Publishing Co., 1989.
- Whitted, Turner, "Managing Geometric Complexity with Enhanced Procedural Models." *SIGGRAPH '86*, Vol. 20, No. 4, (1986), pp. 189–195.

Index

.CPR files, 425, 495
.DES, 255, 278
.RT, files, 393
.SCN files, 136, 177, 227, 505
1024x768x256 Super VGA Mode, 359
2 particle orbits, 80
3 particle orbits, 85, 523
3D Julia Set, 317, 325
3D Mandelbrot Set, 317, 323
3D Strange Attractor, 97

A

AddOns.Inc, 280
AddVertex, 183
affine transformation routines, 37
affine transformations, 178
ambient light, 179, 395, 491
Angl, 62
aspect ratio, 59, 426, 486
attractive cycle period, 288

B

Beads.Dat, 216
bifurcation diagrams, 94, 95
BIOS, 41, 57, 66
blended sky, 289
Bowl.Dat, 295
Box, 485
Bresenham's algorithm, 60

C

camera, 355, 426
Cantor Set, 288
CartesianPlot3D, 63

chaos, 94, 97, 100
checkerboard, 426
ChesPawn.Dat, 216
Circle, 59
ClearPalette, 57
clouds of plasma, 348
color, 491
 and the human eye, 495
 assignments, 230
 histogram, 495
 popularity for display, 495
complex plane, 67, 281
cone, 488
ConePym.Pas, 191
continuous potential method, 323, 325
CosD, 31
CPM2.Dat, 323
CPM.Pas, 323
CPMJ.Pas, 325
CPMJ1.Dat, 325
creating .SCN files, 227
creating objects manually, 188
creating solid model object databases, 183
cross product, 34
CubePlan.Scen, 159
cursor technique, 76
cycle palette, 348
CyclePalette, 59
cylinder, 488
Cylinder.Pas, 195
CylindricalPlot3D, 64

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

D

- Degrees, 31
- dendrite, 288
- Depth, 393, 426, 493
- description file, 278
 - for Model.Pas, 155
- description files for z-Buffer, 257
- DesMake.Pas, 251
- DesMaker.Pas, 155
- diffuse reflection, 179, 491
- display, 495
 - axis, 66
 - color palette, 66
 - modes, 360
 - of a ray traced file, 495
- Display.Pas, 497
- DisplayActualImages, 153
- displaying the height field, 279
- DisplayReflections, 153
- DispObjs.Inc, 138
- distance calculation, 486, 487
- DistEffect, 425
- dot product, 34
- Dragon.Dat, 336
- Dragons, 317
- Draw, 60
- DrawLine3D, 65

E

- edge reflectors for solid model, 134
- elliptic cone, 488
- elliptic paraboloid, 488
- end caps for solid model shapes, 191, 200
- Equation Plot .DAT files, 299
- Equation Plot .SCN files, 159
- equation plots, 207, 299
- Example1.Dat, 395
- Example1.RT, 395
- Example2.Dat, 397
- Example2.RT, 397

- ExitGraphics, 61

- Exp10, 32

- eye, 355

F

- facet arrays, 108
- facet visibility test, 180
- facets, 107
- Fatou dust, 288
- finding nearest object, 486, 487
- FindLmts.Pas, 305
- focal length, 394, 486
- format constraints for .RT files, 393
- FourCols.Scen, 159
- fractal, 67, 91, 317, 323, 325
 - database generators, 317
 - landscape, 317
- Funnels.Dat, 216

G

- geometry of ray tracing, 356
- GetPixel, 66
- Gloss, 179
- Graph.Inc, 41, 42
- Graph2.Inc—High-Resolution, 375
- graphics interface module, 41
- Grid.Dat, 134

H

- Hamilton, Sir William, 330
- Hamilton's Quaternions, 330
- Height Buffer Data Format, 248
- HeightBufferScalingFactor, 245, 248
- hidden facet removal by visibility test, 180
- High Resolution Graphics, 359, 486
- histogram, 495
- HmSphere.Dat, 291
- HmSphere.Pas, 199, 291
- HmToroid.Pas, 295
- HorCol, 426

Horizon Rendering Technique, 241

hyperbolic paraboloid, 488

hyperboloid of one sheet, 488

of two sheets, 488

I

illumination model, 179, 277, 491

InitGraphics, 61

initial conditions for 3 particle orbits, 523

InitNoise, 427

InitPalette1, 58

InitPalette2, 58

InitPerspective, 62

InitPlotting, 62

integer scaling, 108, 248

interpolated pixel buffer, 261

Intersect, 486, 487

intersection test, 486

IntPower, 33

IntSign, 32

IntSqrt, 32

InverseHeightBuffer, 329

iterate an equation, 67, 94, 97, 100, 281,
323, 325, 336

J

Jordan curve theorem, 490

Julia Set, 67, 281, 325

K

keyboard response routines, 232, 251

L

lamps, 395

LgBeads.Dat, 216

light source vector, 160, 179, 229

lightning dendrite, 288

LightPhi, 160, 278

LightTheta, 160, 278

loading an .RT file, 482

LoadInteger, 482

LoadReal, 482

LoadText, 482

local color contribution, 491

LoclWgt, 482

Log, 32

Lorenz Attractor, 100

lunar landscape, 317

M

magnetic fields, 305

MagnFlds.Dat, 305

MagnFlds.Pas, 305

MakeObj.Pas, 188

Mandelbrot Set, 67, 323

Mandelbrot/Julia Set, 77

MapCoordinates, 63, 65

mapping a Julia Set to a Sphere, 288

Marbles.Dat, 400

Marbles.RT, 400

material definitions, 523

materials, 395, 523

Math.Inc, 17

Math2.Inc—High-Resolution, 360

mathematics module, 17

MaxFacet, 115

MaxVertex, 115

MaxWgt, 482

MechPart.Dat, 216

meters, 84, 85

minimum distance object, 486, 487

MinWgt, 482

Mirror, 121

Model.Des, 155, 177

Model.Inc, 107

Model.Pas, 155, 160, 182

Molecule.Scen, 159

Mountain.Dat, 324

Mountan1.Dat, 325

Mountan2.Dat, 325

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

Mountain.Pas, 317
mountains from plasmas, 348
MsetIset.Pas, 67
Multiply3DMatrices, 39

N

Newton, Sir Isaac, 77
noise function, 427
normalize vector, 35

O

object buffer for solid model, 122
object declaration for solid modeling, 121
objects, 395
Observer, 394
ObsPos, 426
ObsRotate, 426
obtaining facet screen coordinates, 181
Office.Dat, 402
Office.RT, 402
Orbit-2P.Pas, 80
Orbit-3P.Pas, 85
orbits, 77
OrientQuadratic, 482

P

PaletteRegister, 57
parametric form of a ray, 487
Park.Dat, 408
Park.RT, 408
ParkNite.RT, 416
particle orbits, 77
Perlin Noise Function, 485
perspective, 62
Phong shading model, 179, 277, 491
pixel buffer for interpolation, 261
Plane.Dat, 118
Plasma, 348
Plasma Mountain (Plasma.Dat), 348
plasma mountains, 348

Plasma.Pas, 348
Plot 41, 359
PlotEqn.Pas, 299
PlotEqns.Pas, 207
plotting equations, 299
Power, 32
Power-Graph ERGO-VGA card, 359
PreCalc, 41
PrepareInvMatrix, 39
PrepareMatrix, 39
Preview Vertices, 152
procedurally defined objects for solid model, 135
projection, 63, 280
PutFacet, 118
PutFacet2—allows reflection for solid model, 181
PutPixel, 66
Pyramid, 485

Q

QSetCons, 336
quadratic surfaces, 488
quadric curve, 488
Quater.Inc, 335, 337
Quater.Pas, 336, 340
quaternion, 330
Quaternion Julia Set (Dragon.Dat), 336
Quaternion Julia Set (RevSolid.Dat), 336
Quaternions, 317
 3D square root, 335
 addition, 335
 multiplication, 330, 335
 squaring, 335
 subtraction, 335
 theory, 330
 uses, 330

R

Radians, 31

Ray Tracing 353

.CPR image files, 425

aspect ratio, 426

color histogram, 495

depth, 426

diffuse reflection, 491

display of a .CPR file, 495

DistEffect, 425

finding nearest object, 486, 487

focal length, 394

fundamentals, 355

initial ray direction, 486

Intersect, 486

intersection tests, 486

Lamps, 395

Local Color Model, 491

material, definitions, 523

material, Checker, 427, 493

material, Grit, 427, 493

material, Marble, 427, 493

material, Sheetrock, 427, 493

material, Smooth, 427, 493

material, Wood, 427, 493

Materials, 395

Objects, 427, 483

Observer, 394

ray-object intersection test, 486

ray-plane intersection, 489

ray-quadratic intersection, 488

ray-sphere intersection, 488

recursion, 493

Reflection, 493

ReflectiveLamps, 425

scene, .RT files, 393

scene, Example1.RT, 395

scene, Example2.RT, 397

scene, Marbles.RT, 400

scene, Office.RT, 402

scene, Park.RT, 408

scene, ParkNite.RT, 416

Shadow Feelers, 491

shape, Box, 485

shape, Circle, 427, 484

shape, Cone, 427, 484

shape, Cylinder, 427, 484

shape, Ground, 427, 483

shape, Lamp, 427, 483

shape, Parallelogram, 427, 483

shape, Pyramid, 485

shape, Quadratic, 427, 484

shape, Ring, 427, 484

shape, Sphere, 427, 484

shape, Triangle, 427, 483

ShootRay, 486

Sky, 426

specular reflection, 491

Textures, 427, 491, 493

ViewDir, 426

ViewTheta and ViewPhi, 426

XPitch and YPitch, 426, 486

ray-object intersection test, 486

ray-plane intersection, 489

ray-quadratic intersection, 488

ray-sphere intersection, 488

RayTrace.Pas, 425

recursion, 91, 336, 348, 426, 493

recursive subdivision of screen, 348

reflected rays, 493

Reflection, 121, 355

reflection screen buffer, 180

reflections for solid model, 153

ReflectiveLamps, 425

ReflWgt, 482

removal of discontinuous zeros, 314

Render.Inc, 243

Render.Pas, 261, 279

Res, 248, 249, 291

revolving 2D profile to generate a solid,
216, 225

RevSolid.Dat, 336

Rocket.Dat, 216

ROM BIOS, 41, 57, 66

Rotate3D, 38

Rx, Ry, Rz, 121

S

scale factors, 108

Scale3D, 38

Scaling, 249

scene definition files for solid modeling, 505

scene files for ray tracing, 393

ScnMaker.Pas, 227, 233

SetMode, 41

SetPalette, 57

shading, 107, 179, 277, 491

shadow feeler, 491

shadows, 355

Shapes.Scen, 159

shininess, 179

ShootRay, 486, 491

ShowGround, 279

ShpMk.Inc, 183

Sierpinski Triangle, 91, 94

Sign, 32

simulation, 77

SinD, 31

Sky, 289, 426

Smooth.Pas, 314

smoothing algorithm—pixel averaging on
subdivision, 348

Solid Model, 152

.SCN file, 136, 177

.SCN file maker, 227

adding objects, 121

color assignments, 230

data file sample, 118

declarations, 108

description file, 155, 177

facet load, 116

facet save, 116

memory requirements, 115

object, Cone, 191

object, Cylinder, 195

object, Equation Plots, 207

object, Hemisphere, 199

object, Pyramid, 191

object, Solids of Revolution, 216

object, Sphere, 203

object, Toroid, 212

object database synthesis, 183

object sorting, 137

procedurally defined objects, 135

reflections, 153

vertex load, 116

vertex save, 116

solid modeling, 105, 107

solid modeling program, 159

solid modeling scene definition files, 505

solid object manual object database entry,
188

solid of revolution, 216, 225

solids of revolution, 317

SolOfRev.Pas, 216

SolOfRev.Scen, 159

Sortable, 121

sorting objects for display in solid model,
137

specular reflection, 179, 491

Sphere.Pas, 203

SphericalPlot3D, 65

SphrPix, 288

SphrWall.Scen, 159

spread of normal calculation for fractal ob-
jects, 256, 277 336

stacked spheres from procedure, 160

StakTors.Scen, 159

stats, 482

STB Systems, Inc., 359

strange attractors, 97

surface normal vectors, 179, 277

Swap, 59

Sx, Sy, Sz, 121

T

texture, 427, 491

Three Particle Orbits, 523

three-dimensional Julia Set, 325

three-dimensional Mandelbrot Set, 323

three-dimensional plotting, 62

Tile, 426

Tilt, 62, 278

Title, 61

toroid, 212

Toroid.Pas, 212

torus, 212

Transform, 40

Translate3D, 37

Tseng Laboratories, 359

turbulence, 427

Tx, Ty, Tz, 121

types of Julia set, 288

U

UnVec, 34

UnVecInt, 34

V

Vec, 33

VecAdd, 36

VecAdd3, 36

VecAddScalMult, 36

VecCopy, 36

VecCross, 34

VecDot, 34

VecElemMult, 37

VecInt, 33

VecLen, 35

VecLinComb, 36

VecMatxMult, 35

VecNormalize, 35

VecNull, 37

VecNullInt, 37

VecScalMult, 36

VecScalMultInt, 36

VecSub, 35

VecSubInt, 35

vector and matrix routines, 33

vertex arrays, 108

VGA, 58, 495

view, 160, 229, 278, 355, 426, 486

view direction, 62

view distance, 63

view vector, 160, 179

ViewDir, 426, 486

ViewPhi, 160, 278, 426

ViewSlic.Pas, 310

ViewTheta, 160, 278, 426

ViewVec, 486

visibility test, 180

W

WaitForKey, 61

Watt, Alan, 485

weights, 482

Well.Scen, 159

Wire frame, 105, 152

WireFrameFacet, 118

X

XPitch, 426, 486

XRes, 61

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL

Y

y offset, 278
YPitch, 426, 486
YRes, 61

Z

Z-buffer, 241

z-buffer

- add ons for backgrounds, 280
- databases and their generation, 291
- description file maker, 251
- description files, 257, 278
- display of height field, 279
- finding z limits, 305
- fractal database generators, 317
- height buffer scaling, 248
- loading height buffer, 248
- projection, 280
- quaternion generation program, 336
- removal of discontinuous zeros, 314
- rendering program, 261
- Res, 248
- saving height buffer, 248
- spread of normal calculation, 256
- viewing a slice of the height buffer, 310

z-buffer database

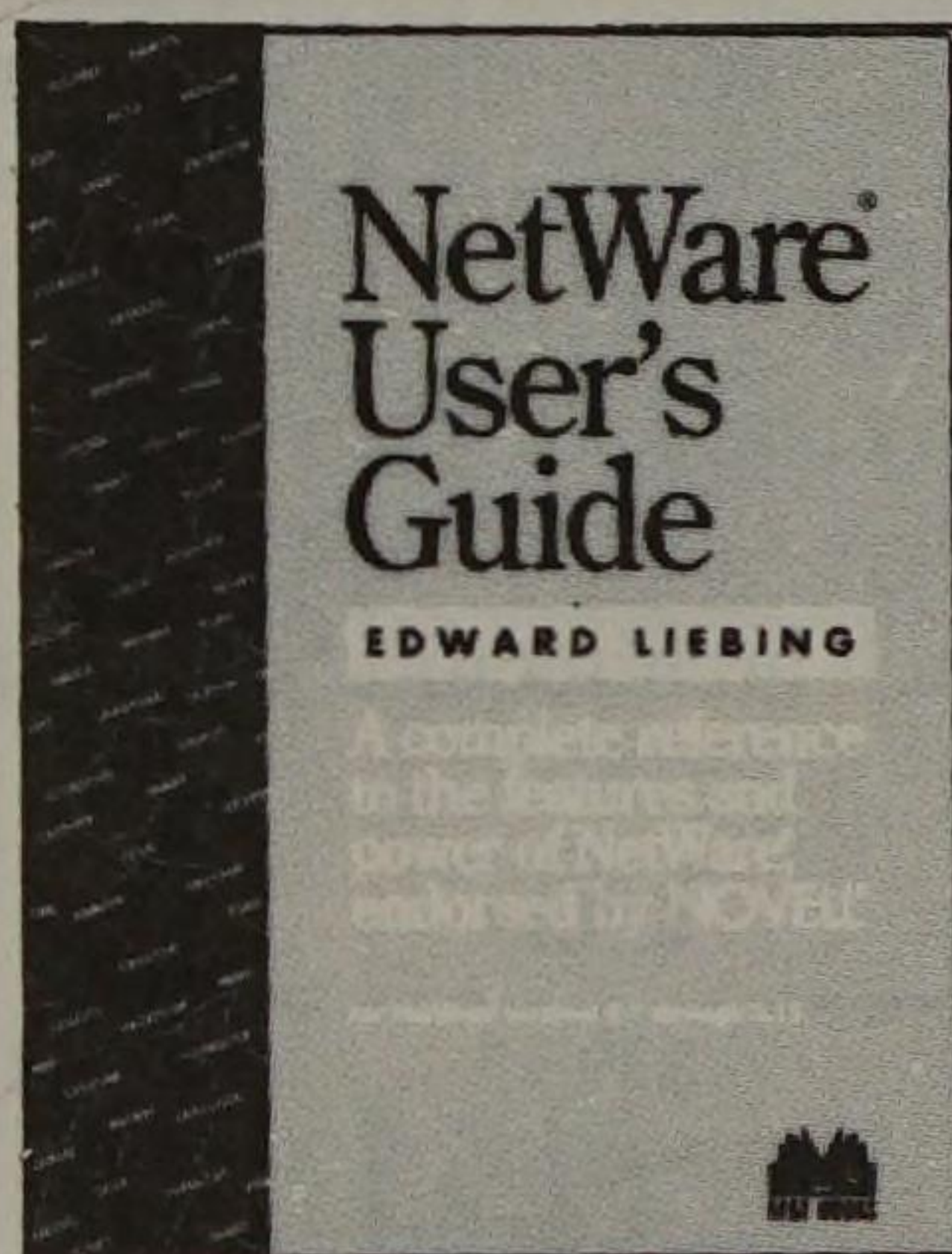
- 3D Julia Set by continuous potential, 325
- 3D Mandelbrot Set by continuous potential, 323
- dragon quaternion, 336
- equation plots, 299
- fractal mountains, 317
- hemisphere, 291
- hemitoroid, 295
- magnetic field representation, 305
- plasma mountains, 348
- solid of revolution quaternion, 336
- structure, 243

z-buffering theory, 243

ZenCol, 426

ZeroMatrix, 37

A Library of Technical References from M&T Books



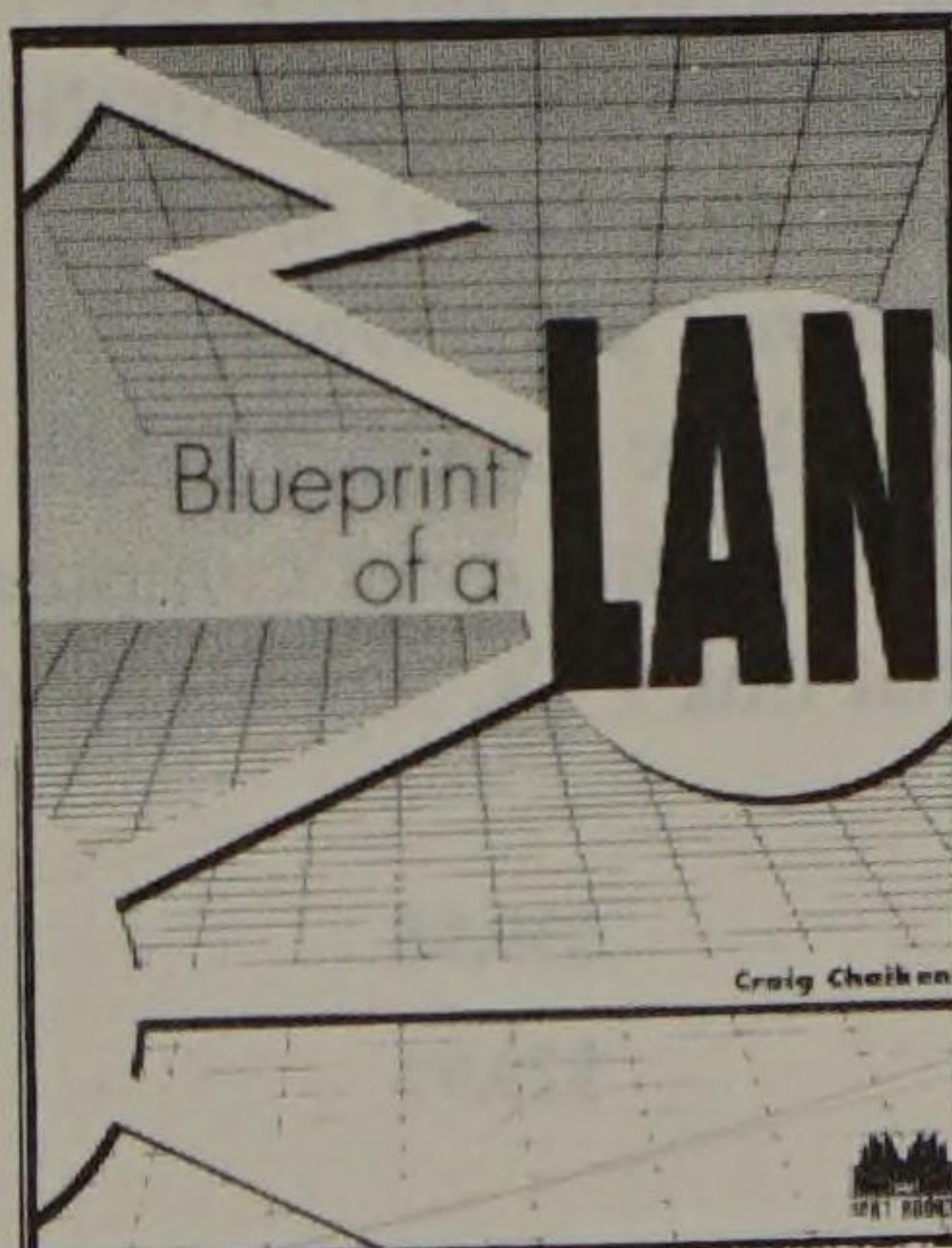
NetWare User's Guide by Edward Liebing

Endorsed by Novell, this book informs NetWare users of the services and utilities available, and how to effectively put them to use. Contained is a complete task-oriented reference that introduces users to NetWare and guides them through the basics of NetWare menu-driven utilities and command line utilities. Each utility is illustrated, thus providing a visual frame of reference. You will find general information about the utilities, then specific procedures to perform the task in mind. Utilities discussed include NetWare v2.1 through v2.15. For advanced users, a workstation troubleshooting section is included, describing the errors that occur. Two appendixes, describing briefly the services available in each NetWare menu or command line utility are also included.

Book only

Item #071-0

\$24.95



Blueprint of a LAN by Craig Chaiken

Blueprint of a LAN provides a hands-on introduction to micro-computer networks. For programmers, numerous valuable programming techniques are detailed. Network administrators will learn how to build and install LAN communication cables, configure and troubleshoot network hardware and software, and provide continuing support to users. Included are a very inexpensive zero-slot, star topology network, remote printer and file sharing, remote command execution, electronic mail, parallel processing support, high-level language support, and more. Also contained is the complete Intel 8086 assembly language source code that will help you build an inexpensive to install, local area network. An optional disk containing all source code is available.

Book & Disk (MS-DOS)

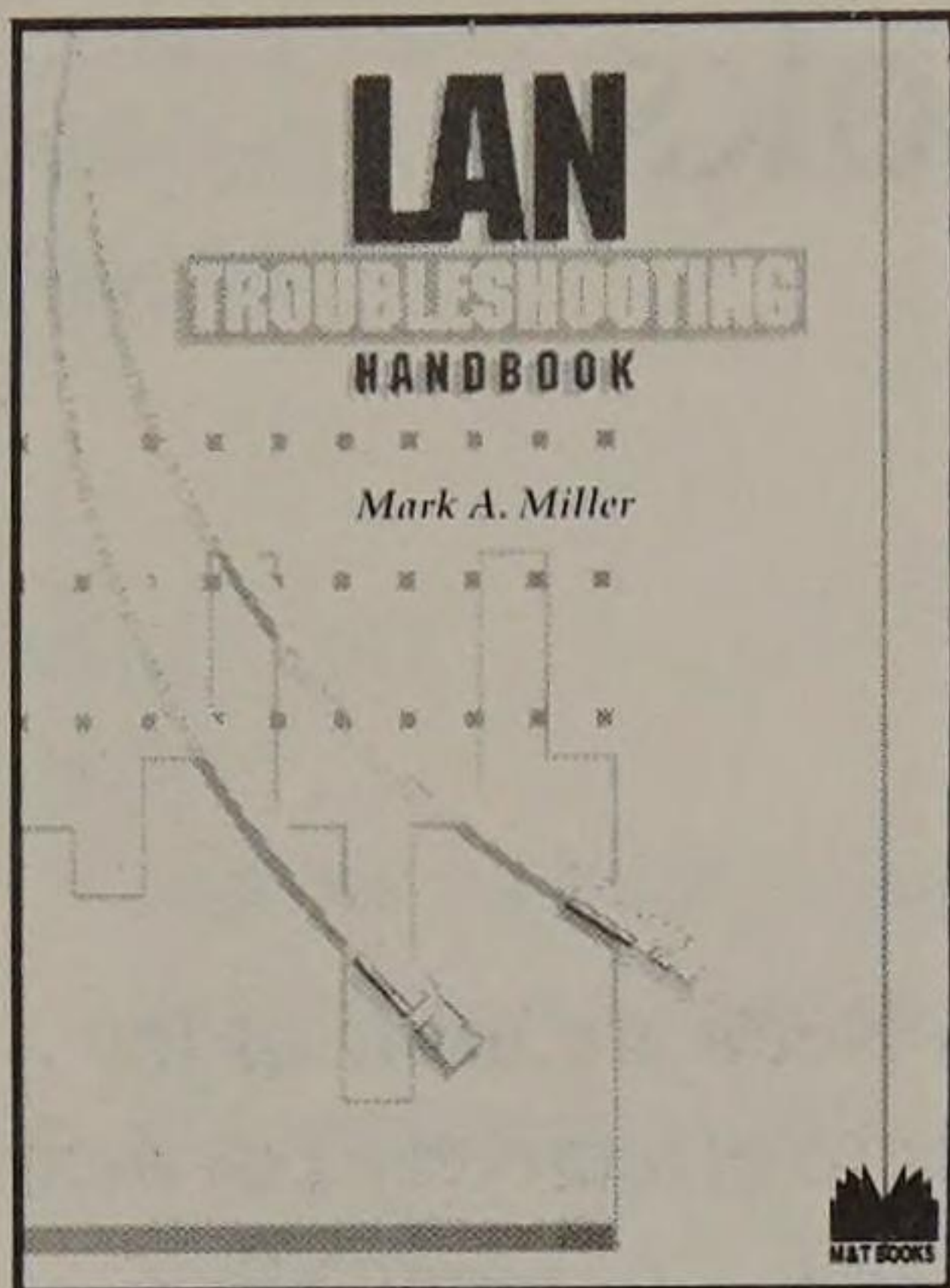
Item #066-4

\$39.95

Book only

Item #052-4

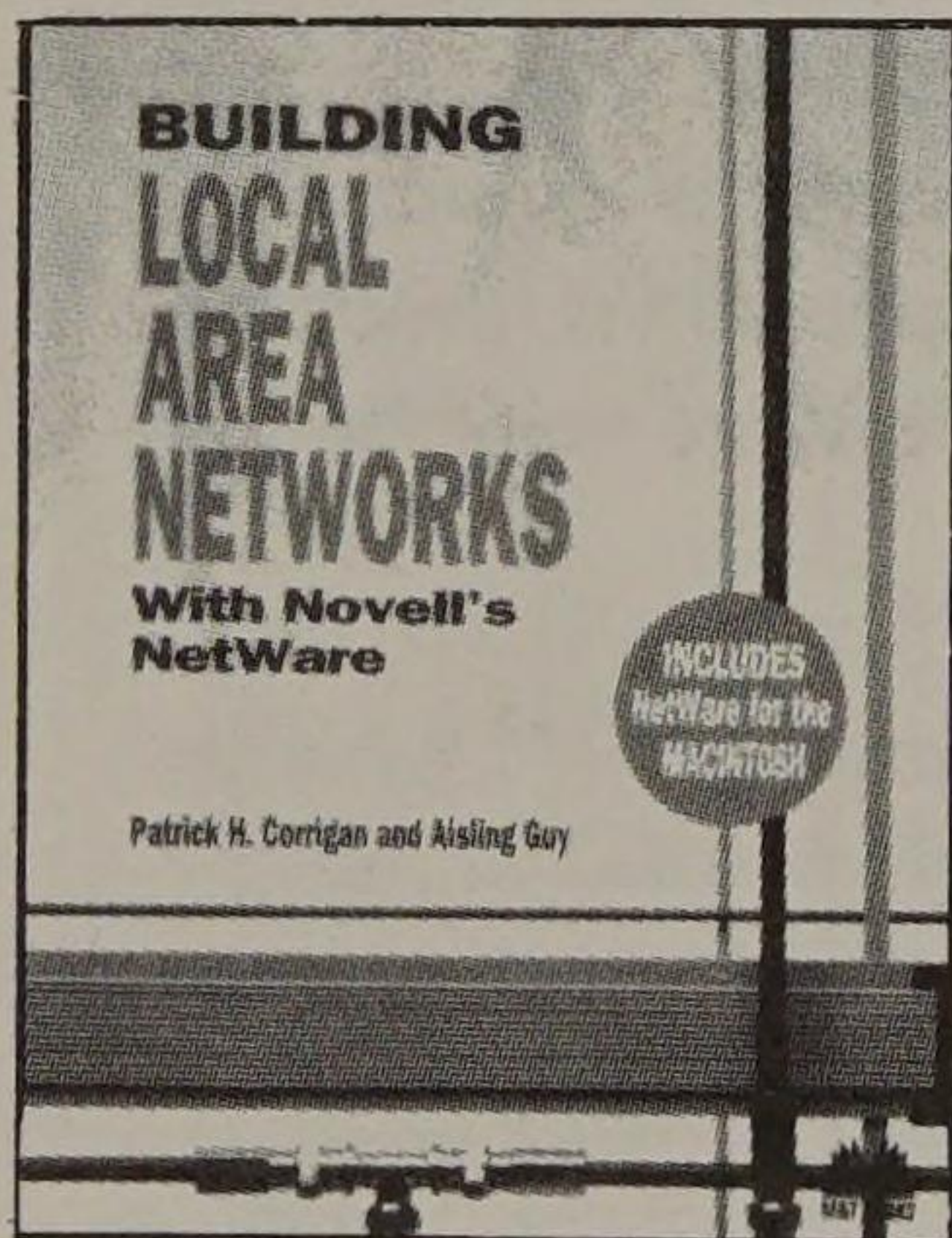
\$29.95



LAN Troubleshooting Handbook by Mark A. Miller

This book is specifically for users and administrators who need to identify problems and maintain a LAN that is already installed. Topics include LAN standards, the OSI model, network documentation, LAN test equipment, cable system testing, and more. Addressed are specific issues associated with troubleshooting the four most popular LAN architectures: ARCNET, Token Ring, Ethernet, and StarLAN. Each are closely examined to pinpoint the problems unique to its design and the hardware. Handy checklists to assist in solving each architecture's unique network difficulties are also included.

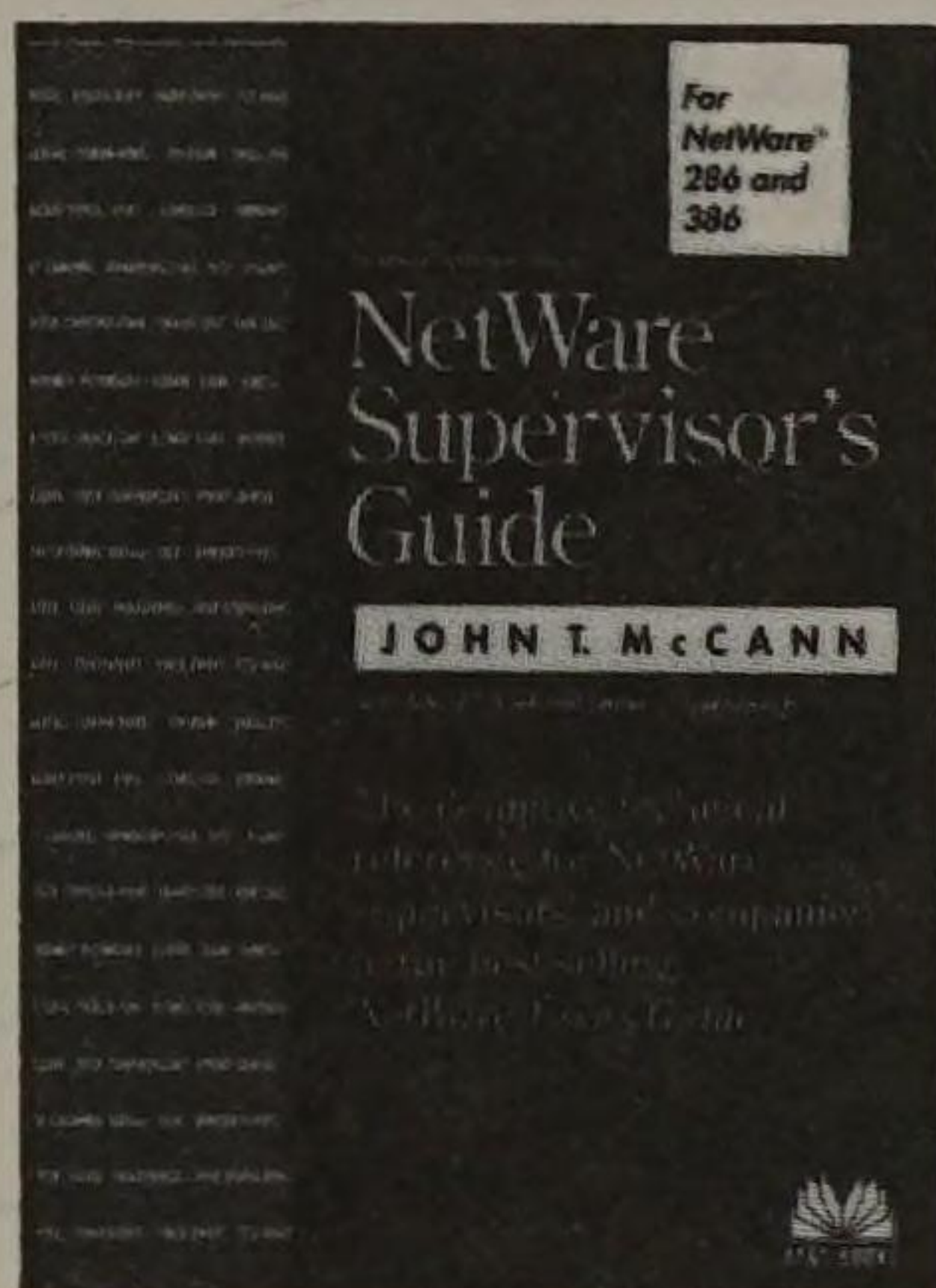
Book & Disk (MS-DOS)	Item #056-7	\$39.95
Book only	Item #054-0	\$29.95



Building Local Area Networks with Novell's NetWare by Patrick H. Corrigan and Aisling Guy

From the basic components to complete network installation, here is the practical guide that PC system integrators will need to build and implement PC LANs in this rapidly growing market. The specifics of building and maintaining PC LANs, including hardware configurations, software development, cabling, selection criteria, installation, and on-going management are described in a clear "how-to" manner with numerous illustrations and sample LAN management forms. *Building Local Area Networks* gives particular emphasis to Novell's NetWare, Version 2.1. Additional topics covered include the OS/2 LAN manager, Tops, Banyan VINES, internetworking, host computer gateways, and multisystem networks that link PCs, Apples, and mainframes.

Book & Disk (MS-DOS)	Item #025-7	\$39.95
Book only	Item #010-9	\$29.95



NetWare Supervisor's Guide

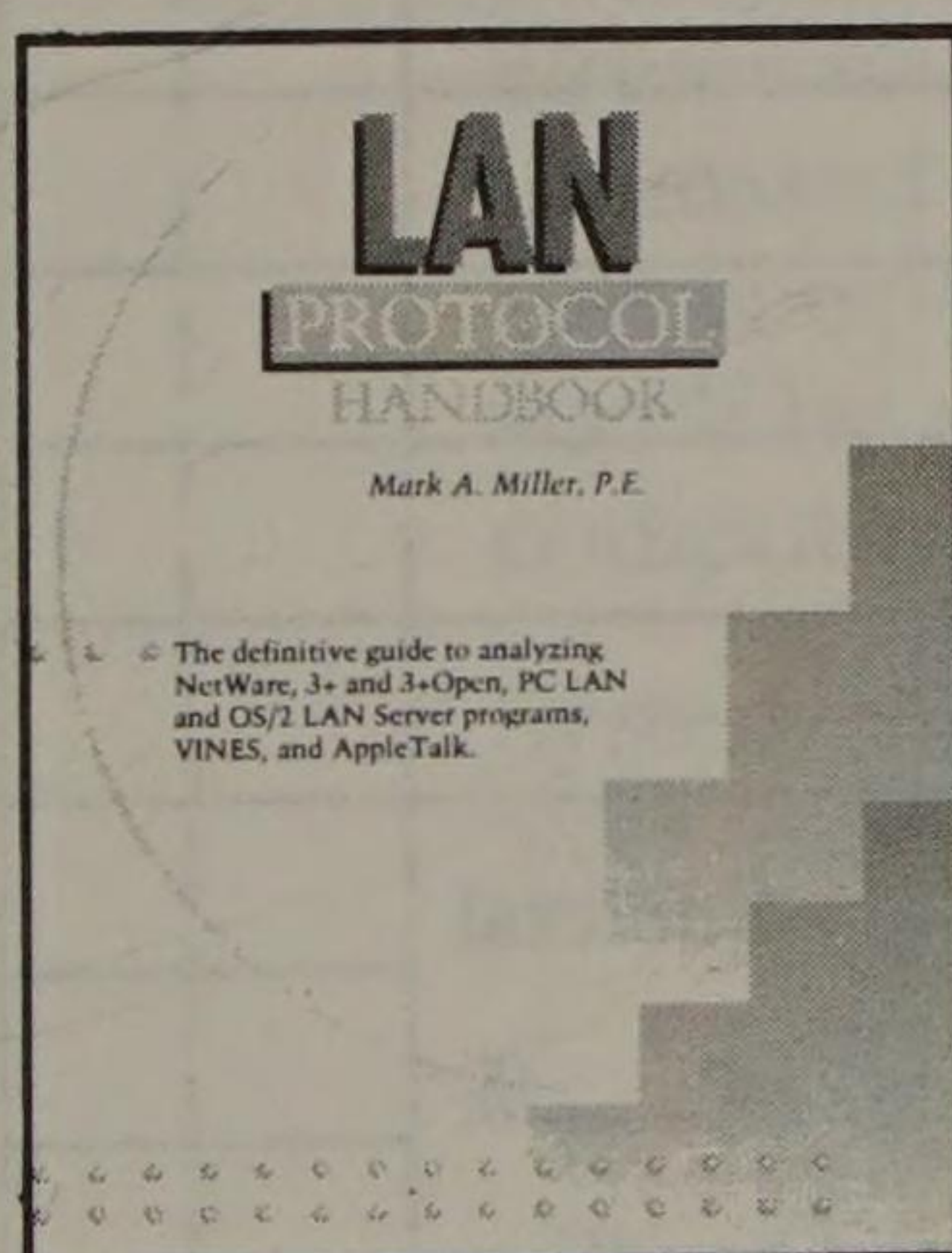
by John T. McCann, Adam T. Ruef, and Steven L. Guengerich

Written for network administrators, consultants, installers, and power users of all versions of NetWare, including NetWare 386. Where other books provide information on using NetWare at a workstation level, this definitive reference focuses on how to administer NetWare. Contained are numerous examples which include understanding and using NetWare's undocumented commands and utilities, implementing system fault tolerant LANs, refining installation parameters to improve network performance, and more.

Book only

Item #111-3

\$29.95



LAN Protocol Handbook **by Mark A. Miller, P.E.**

Requisite reading for all network administrators and software developers needing in-depth knowledge of the internal protocols of the most popular network software. It illustrates the techniques of protocol analysis—the step-by-step process of unraveling LAN software failures. Detailed are how Ethernet, IEEE 802.3, IEEE 802.5, and ARCNET networks transmit frames of information between workstations. From that foundation, it presents LAN performance measurements, protocol analysis methods, and protocol analyzer products. Individual chapters thoroughly discuss Novell's NetWare, 3Com's 3+ and 3+Open, IBM Token-Ring related protocols, and more!

Book only

Item 099-0

\$34.95

ORDER FORM

To Order:

Return this form with your payment to M&T books, 501 Galveston Drive, Redwood City, CA 94063 or call toll-free 1-800-533-4372 (in California, call 1-800-356-2002).

ITEM #	DESCRIPTION	DISK	PRICE
Subtotal			
CA residents add sales tax ____%			
Add \$3.50 per item for shipping and handling			
TOTAL			

Charge my:

- ☐ Visa
☐ MasterCard
☐ AmExpress

☐ Check enclosed,
 payable to
M&T Books.

CARD NO. _____

SIGNATURE _____

EXP. DATE _____

NAME _____

ADDRESS _____

CITY _____

STATE _____

ZIP _____

M&T GUARANTEE: If your are not satisfied with your order for any reason, return it to us within 25 days of receipt for a full refund. Note: Refunds on disks apply only when returned with book within guarantee period. Disks damaged in transit or defective will be promptly replaced, but cannot be exchanged for a disk from a different title.

Order the *Advanced Graphics* *Programming in Turbo Pascal* Source Code Disk

No need to manually type in the book's source code. This optional disk (PC/MS-DOS format) contains all the source code for the programming examples listed in the book. You'll find everything from tips and techniques to complex programming examples. The program for each display discussed is listed so you can use it in its existing form or modify it to suit your own requirements. After trying out the programs in this book, you'll be amazed at the graphics capabilities that lay hidden in your computer!

System requirements: IBM/PC-compatible computer with a VGA card and VGA monitor. Need Turbo Pascal version 5.5 or later.

To order, return this postage-paid card with your payment to:
M&T Books,

501 Galveston Drive, Redwood City, CA 94063.

Or call **TOLL FREE 1-800-533-4372 (in CA 1-800-356-2002).**

☐ **Yes!** Please send me the optional source code disk \$20.00
CA residents add applicable sales tax %
Total

☐ Send me your latest M&T Books Direct Catalog.

☐ Check enclosed. Make payable to M&T Books.

Charge my ☐ VISA ☐ MC ☐ AmEx

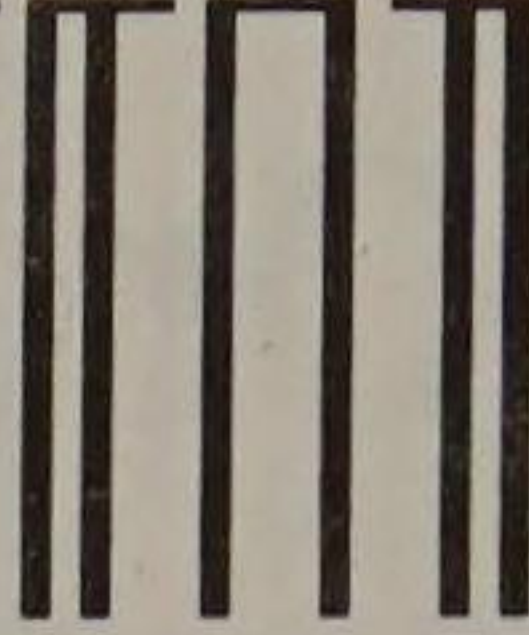
Card No. Exp. Date

Name

Address

City State Zip

Note: Disks damaged in transit may be returned for a replacement. No credit
or refund given.



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

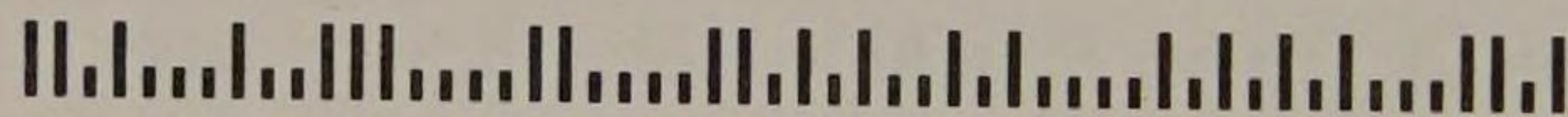
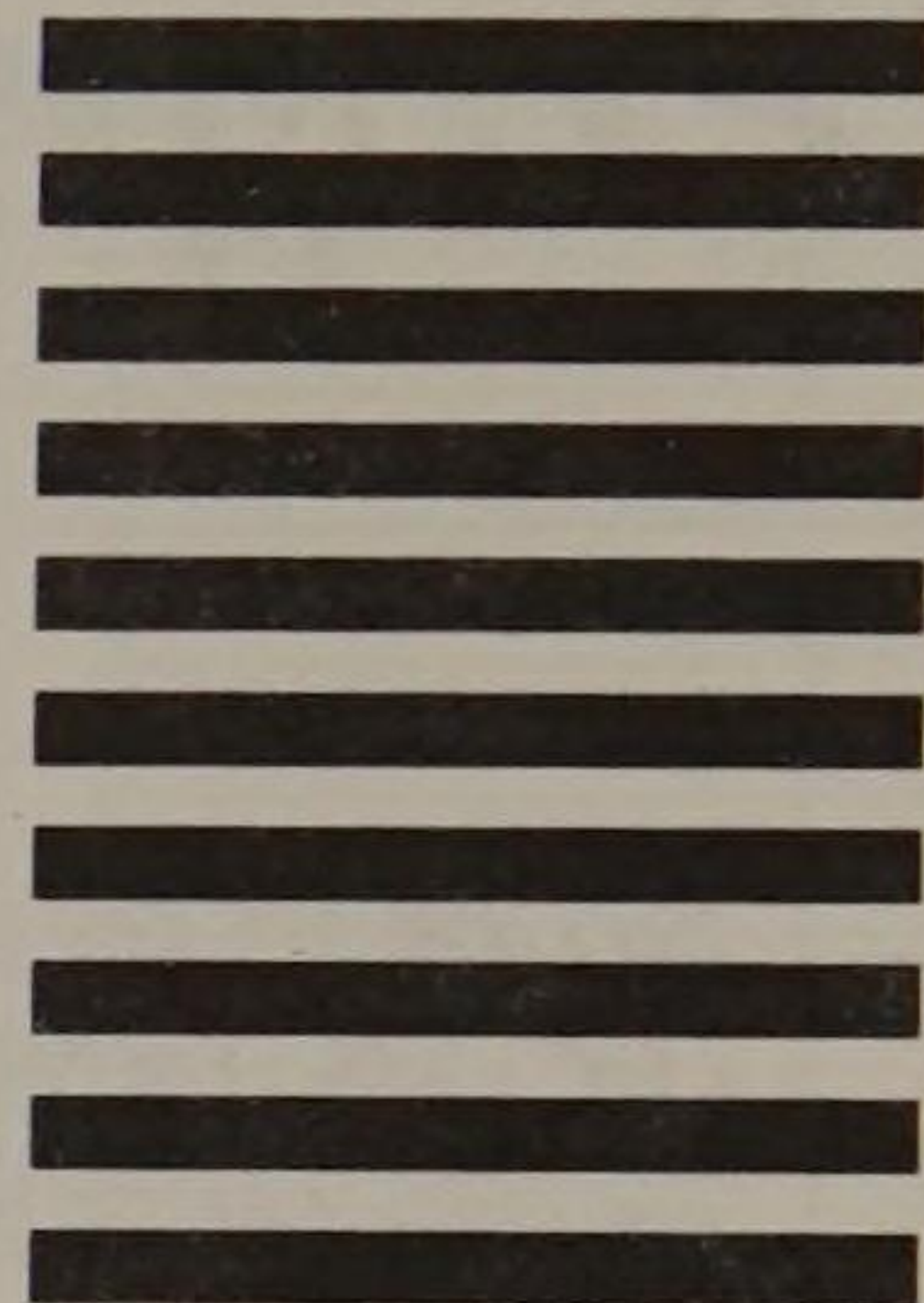
BUSINESS REPLY MAIL

FIRST CLASS MAIL PERMIT 871 REDWOOD CITY, CA

POSTAGE WILL BE PAID BY ADDRESSEE

M&T BOOKS

501 Galveston Drive
Redwood City, CA 94063-9929



PLEASE FOLD ALONG LINE AND STAPLE OR TAPE CLOSED

ADVANCED GRAPHICS PROGRAMMING IN TURBO PASCAL



Advanced Graphics Programming in Turbo Pascal™ details the fundamentals of graphics programming for the IBM PC and compatibles, teaching Turbo Pascal programmers of all levels how to create compelling graphic images easily and efficiently. Through detailed discussions and sample programs you'll gain the tools, techniques, and know-how to draw and fill geometric shapes; design three-dimensional figures; generate the Mandelbrot set; and much more.

Topics include:

- **Universal Routines.** Understanding mathematical functions, working with the graphics interface module, and learning how to use the modules through sample programs.
- **Wire Frame and Solid Modeling.** Defining solid modeling theory and database structure, using the solid modeling program, exploring the description file maker, generating object databases, and creating and displaying scene files.
- **Z-Buffering and Horizon Rendering.** Examining z-buffering theory, database structure, and the horizon rendering method; using the z-buffer rendering program; and exploring fractal programs to generate z-buffer databases.
- **Ray Casting and Ray Tracing.** Understanding ray tracing theory and database structure, developing the ray tracing program, and using database generators for ray tracing programs.

Packed with examples and sample programs, this book provides the information you need to explore the exciting field of computer graphics. And because it contains all of the source code, you can easily modify the functions to suit your specific needs. All of the listings are available on disk in MS/PC-DOS format. The programs require an IBM PC or compatible with a VGA card, a VGA monitor, and Turbo Pascal 5.5 or better.

Roger T. Stevens is a seasoned graphics programmer. He is the author of the best-selling *Fractal Programming and Ray Tracing with C++*, *Advanced Fractal Programming in C*, *Fractal Programming in Turbo Pascal*, *Fractal Programming in C*, and *Graphics Programming in C*. **Christopher D. Watkins** is the founder of his own research and engineering company located in Atlanta, Georgia. Fields of research and development include computer graphics and simulation, automation and control, electronics, and electrical engineering.

Why this book is for you – See page 1



M&T Books
501 Galveston Drive
Redwood City, CA 94063



ISBN 1-55851-131-8
>\$29.95